Community Experience Distilled

HTML5 Game Development with ImpactJS

A step-by-step guide to developing your own 2D games

Foreword by Dominic Szablewski, creator of ImpactJS

Davy Cielen
Arno Meysman

[PACKT] PUBLISHING

HTML5 Game Development with ImpactJS

A step-by-step guide to developing your own 2D games

Davy Cielen

Arno Meysman

HTML5 Game Development with ImpactJS

First published: March 2013

Production Reference: 1140313

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-456-8

www.packtpub.com

Cover Image by Shutterstock

Credits

Authors
Davy Cielen
Arno Meysman

Reviewers
Makzan
Kevin Roast

Acquisition Editor
Robin de Jongh

Lead Technical Editor
Arun Nadar

Technical Editors
Soumya Kanti
Veena Pagare

Copy Editors
Alfida Paiva
Laxmi Subramanian
Ruta Waghmare
Aditya Nair

Project Coordinator
Anugya Khurana

Proofreader
Bernadette Watkins

Indexer
Hemangini Bari

Production Coordinator
Arvindkumar Gupta

Graphics
Valentina Dsilva

Cover Work
Arvindkumar Gupta

Foreword

If you are new to programming, let me congratulate you for choosing one of the best ways to learn it.

Writing games can be way more fun and more rewarding than playing them. It will keep you on the edge of your seat when you first make something that you can move on the screen, when you first make something that resembles a game, when you first make something that is fun to play, and when you first put something out on the Internet for others to enjoy.

You will soon discover that programming is no black magic, but just a bunch of simple concepts glued together.

If you are new to JavaScript, the syntax can be quite confusing and some of its concepts may seem alien at first. But hidden under all that "wait, what?" moments is a language that is actually quite powerful, elegant, and concise. Stick to the simple stuff at the beginning, but don't be too afraid to get your feet wet. If you come back to something you wrote a month ago, you will have discovered more explicit and easier ways to do it. This is true for every language, but the learning curve of JavaScript is an especially rewarding one.

If you are new to game development, you will see that it's no more difficult than any other kind of development. Once you get the idea of a game loop in your head and understand how objects are moved over time, everything else will suddenly make sense.

One of the biggest revelations when I started to make games was how almost comically fast computers are. The amount of things you can do in 16 milliseconds, 60 times per second, is truly astonishing.

At the same time, it's astonishing how easily you can end up doing huge amounts of operations. Compare 100 objects with each other? That's 10,000 comparisons! Still, you can do such things and much more. So don't be afraid to try things that you think may be too slow. Your game will most likely not be bound by computation performance.

If you are new to Impact and I would think you are, since you bought this book, you will find that it is just the aforementioned bunch of simple concepts, neatly arranged for you.

One of Impact's most important aspects is its simplicity. I took great care to make the API, the functionality of Impact that is exposed to you as logical, consistent, and easy to grasp as possible. There are no callbacks anywhere; everything that's happening is happening in the game loop and it's happening in sequence. This makes it easy to reason about the state of your game at all times.

But this API shouldn't discourage you to dig deeper. In fact, I would like to invite you to take a closer look at Impact's source code.

Let's be clear here. Almost every time I had a look at the source code of a large software library or application, I felt quite lost. If I finally found the function I was interested in, it turned out to be just a stub with three lines of code that calls the actual implementation elsewhere, which in turn just calls five different functions with nondescript names in some other files.

Impact is not that large a software library. It's a very concise framework, you don't have to read through thousands of pages of code scattered over hundreds of files. Most things are self contained and you don't have to jump around a lot to get an understanding of what a certain function does.

No black magic involved.

Dominic Szablewski

Creator of ImpactJS

About the Authors

Davy Cielen is the co-owner of An Ostrich On Mars, a graphic design and marketing agency with a special branch of game design, graphics, and game development. He has a background in analytics, marketing, and mathematics. Davy is seriously in love with game design and web technologies.

I want to thank my wife Filipa for the support she has given me. This book has cost Arno and me a lot of time. Filipa, you have always been there to support us and always push me to grow as a husband and as a person.

A special thanks to my father, Nico Cielen, who passed away recently. He has always guided me in the journey of life. He is a great man. Jeanine, I will never forget how you have taken care of him.

I also want to credit my family, friends, and colleagues for their support and understanding while we developed this book. It has consumed a lot of my time, but I could always count on your support.

Last but not the least, I want to thank Arno for the hard work he has put into this book and the people at Packt Publishing for the opportunity they have given us to write this.

Arno Meysman is the co-owner of An Ostrich On Mars, a graphic design and marketing agency with a special branch of game design, graphics, and game development.

Arno Meysman is a specialist at customer and web analytics using statistics and has always been very interested in game development, including graphical design. He started using the ImpactJS engine for hobby projects when it was first released in 2010.

I would like to give thanks to my family and friends who have supported me all the way through the process of writing this book. It has not always been easy to stay at home writing, while I could be out partying and discovering new things. I want to give special thanks to my parents, brother, and girlfriend for always being there for me, regardless of what crazy plans I come up with and execute. A big thank you to all the people of PacktPub involved in the process of making this book for guiding us all the way through. Last but not least I would like to thank my co-author Davy. Without his technical support, writing this book would have been near impossible.

About the Reviewers

Thomas Mak, also known as Makzan, is a developer who specializes in web development and game design. He has over 10 years of experience in building digital products, including real-time multiplayer interaction games and iOS applications.

He is currently a founder of a game development company, 42games (`http://42games.net`), where he makes game-development tutorials and online learning resources.

He wrote two books and one screencast series for building a Flash virtual world and making games with HTML5 and related web standards.

I would like to thank my family and my wife, Candy Wong, for supporting all my writings.

Kevin Roast is a frontend software developer with 15 years, professional experience and a lifelong interest in computer science and computer graphics. He has developed web software for several companies and is a founding developer at Alfresco Software Ltd. He is very excited by the prospect of the HTML5 standardization of the Web and the progress of web-browser software in recent years. He was the co-author of a book, *Professional Alfresco: Practical Solutions for Enterprise Content Management*, and has been a technical reviewer on several HTML5 and development related books.

I would like to thank my wife for putting up with me tapping away in the evenings reviewing book chapters, and to my three kids Ben, Alex, and Izzy for being little funsters.

Table of Contents

Preface 1

Chapter 1: Firing Up Your First Impact Game 7

Installing a XAMPP server 7

Installing the game engine: ImpactJS 10

Choosing a script editor 12

Running the premade game 12

Debugging your game with the browser and ImpactJS 13

Debugging with the browser 15

Firebug – the Firefox debugger 15

Chrome debugger 16

Exercises in debugging with Chrome and Firebug 17

Debugging with ImpactJS 22

Which helpful tools are out there 25

Ejecta 25

AppMobi 25

PhoneGap 27

lawnchair 27

Scoreoid 28

Playtomic 29

one.com webhost 29

Summary 31

Chapter 2: Introducing ImpactJS 33

Building your own levels 34

Creating, loading, and saving levels in Weltmeister 34

Layers and the z axis 36

Adding and removing entities and objects 39

The collision layer 41

Connecting two different levels 42

Objects – playable and non-playable characters **46**
The ImpactJS entity 46
Setting up player controls 51
Position, acceleration, and speed 55
The game's graphics: sprites and animation sheets 58
Spawning, health, and death 60
The camera view **63**
The game canvas 63
Free to move camera 64
Automatically following camera 65
Adding music and sound effects **66**
Playing background music 66
Introducing sound effects 67
Game physics with Box2D **68**
Gravity and force 69
Collision impact and bounciness 70
Summary **71**
Chapter 3: Let's Build a Role Playing Game 73
The RPG game setting **74**
Building an RPG level **75**
Adding a playable character **80**
Introducing a defeatable opponent **84**
Giving the player some weapons **87**
Spawning a projectile 87
Causing harm with a projectile 91
Bringing your NPCs to life with artificial intelligence **93**
The NPC's behavior 94
The NPC's decision making process 97
Pickup items to help out your player **101**
Healing your player with a potion 101
Becoming rich with coins 102
Keeping score for player feedback **103**
Transitioning from one area to another **106**
NPCs and conversation **108**
The speech balloon 108
Adding a talking non-playable character 112
The final battle **114**
Summary **116**

Chapter 4: Let's Build a Side Scroller Game **117**
The side scroller game setting **118**
Preparing the game for Box2D 119
Building a side scroller level **120**
The playable character **122**
Adding a minor foe **127**
Introducing formidable weapons **129**
Shooting a projectile 129
Building an actual bomb 133
Artificial intelligence **138**
Pickup items **140**
Adding a normal crate 140
Implementing a healing crate 142
Keeping score **144**
Transitioning from one level to another **145**
The final battle **148**
Summary **152**
Chapter 5: Adding Some Advanced Features to Your Game **153**
The Start and Game-over screen **154**
The game's Start screen 154
The Victory and Game-over screens 156
More advanced debugging options 159
Short introduction to unit testing 159
Adding your own debug panel to the ImpactJS debugger 161
Handling game data **165**
Adding data to your game 165
Using cookies to store data on the player's computer 168
Local storage 173
Using lawnchair as a versatile way of storing data 174
Extra functionalities for the RPG **177**
Moving the player with a mouse click 177
Adding intelligent spawn locations 182
Introducing basic conversation 185
Adding a basic Head-Up Display 193
Artificial intelligence: The hive mind **197**
Implementing Playtomic for game analytics **200**
Summary **205**

Chapter 6: Music and Sound Effects 207
Making or buying the music 208
Buying tunes and sound effects 208
Making a basic tune using FL Studio 210
Adding background music to your game 216
Playing a sound when an action has happened 217
Tips when using sound files in your game 219
Summary 219
Chapter 7: Graphics 221
Making/buying the graphics 222
Where to buy graphics 223
Introduction to vector graphics 224
Creating your own avatar using Adobe Photoshop 230
Adding your creation to the RPG 232
Hints for graphics in HTML5 233
Summary 234
Chapter 8: Adapting Your HTML5 Game to the Distribution Channels 235
Preparing your game for the web browser 236
Preparing our game for mobile web browsers 238
Turning the game into a web app for the Google Chrome Web Store 243
Pushing your game to Android's Google play store 247
Making your game available on Facebook 249
Direct canvas game acceleration with AppMobi 251
Summary 255
Chapter 9: Making Money with Your Game 257
Your game development strategy 257
Making money with game apps 261
Google Play 262
iOS App Store 264
The Chrome Web Store 265
In-game advertising 266
Selling distribution rights with MarketJS 270
Summary 272
Index 275

Preface

You are here today reading this book because you want to make video games. You wish to build your own video game, which can be run both in people's browsers as well as on their smartphones and tablets. All of this is possible today, though it has not been for the longest time. The reason you can now build your own game with relative ease is two fold: **HTML5** and **ImpactJS**.

HTML5 is the latest version of our World Wide Web Hyper Text Markup Language and is the universal language for web pages. HTML has been around since the beginning of the 1990s when a CERN (the European organization for nuclear research) employee by the name of Tim Berners-Lee first introduced it. New versions were released quite rapidly: Version 2 in 1995, Version 3 in 1997, and Version 4 later that year. We had been using roughly the same version of HTML for over a decade until finally HTML5 saw the light in 2008. With the growing need for multimedia implementation, companies had been building browser plugins to play music, show movies, and so on. The Flash player is probably one of the best known plugins on this front. As a game developer you can still opt for using Flash and ActionScript, but we don't know how long Flash still has until (if ever) it will be fully replaced by HTML5. What will happen to Flash games remains hard to predict but one thing is pretty certain: the future of HTML5 based games looks bright. Since HTML5 emerged, browsers have been slowly increasing compatibility for it. HTML5 is a huge step forward as it introduces new elements to allow playing music and videos on the web pages themselves.

However, the most important new feature for us is the introduction of the `<canvas>` element. The `<canvas>` element is basically a placeholder for your graphical elements to appear on. Together with the use of JavaScript, it became possible to build browser games outside of Flash players. However, JavaScript in itself is not geared towards building games. In its raw form, you would be able to use it to build a game but it would prove dauntingly difficult. Thus, the last necessary ingredient is a JavaScript library with the sole purpose of game development. This is where ImpactJS comes into play.

ImpactJS is in essence a library of JavaScript code capable of making your life as a game developer a lot easier. ImpactJS was developed by Dominic Szablewski, a German genius, to say the least. The ImpactJS game engine has the advantage of enabling you to build a game very quickly with only a basic knowledge of JavaScript and HTML. This allows even a newbie programmer to focus on what they love: building the actual game. ImpactJS also comes with a very intuitive level editor and debug system, which we will also get into during the course of this book. ImpactJS is designed to build tile-based two-dimensional games. For instance, if you like to build a side-scrolling game such as **Mario** or a top-down game such as **Zelda**, you will want to go with ImpactJS. Now without much further ado, let's dive into the action by moving on to *Chapter 1*, *Firing Up Your First Impact Game*, where we will prepare ourselves for game development by gathering the necessary resources for it.

What this book covers

Chapter 1, *Firing Up Your First Impact Game* helps us to set up our development environment, get our first game up and running, and take a look at a few useful tools for the HTML5 game developer.

Chapter 2, *Introducing ImpactJS* dives into the basics of ImpactJS by exploring some of its key concepts using a premade game.

Chapter 3, *Let's Build a Role Playing Game* is a guide to building a top-down game from the ground up.

Chapter 4, Let's Build a Side-Scroller Game helps us build a side-scroller game from scratch, making use of the Box2D physics engine.

Chapter 5, Adding Some Advanced Features to Your Game teaches us to add some advanced features such as advanced artificial intelligence and data storage to the RPG game that we built in *Chapter 3, Let's Build a Role Playing Game*.

Chapter 6, Music and Sound Effects takes us deeper into how to use music and sound effects in ImpactJS, where to buy them, and how to make a basic tune with FL Studio.

Chapter 7, Graphics teaches us to create both vector and Photoshop graphics and explore the option of buying them from artists and specialized websites. Making your own graphics or buying them elsewhere is an important trade-off.

Chapter 8, Adapting Your HTML5 Game to the Distribution Channels helps us take a look at a few of the options for deploying your game to the different devices out there and how it can be done technically. This is the final step of the game development process.

Chapter 9, Making Money with Your Game takes a look at a few of the options to make money as a game developer, from taking care of your own sales and marketing, to selling your distribution rights.

What you need for this book

Following are the software requirements for executing the code given in the book:

- Server (example: XAMPP). Free to download.
- JavaScript code editor (example: Komodo edit). Free to download.
- ImpactJS game engine. Buy at `www.impactjs.com`.
- Google Chrome browser. Free to download.
- Firefox browser and Firebug plugin. Free to download.
- FL Studio. Not free but only relevant for *Chapter 6, Music and Sound Effects*.
- Photoshop. Not free but only relevant for *Chapter 7, Graphics*.
- Inkscape. Free to download.

Who this book is for

This book is for anyone with at least a basic knowledge of JavaScript, CSS, and HTML. If you have the desire to build your own game for your website or an app store but have no idea how and where to begin, this book is for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Open your browser and type `localhost` in the address bar."

A block of code is set as follows:

```
EntityPlayer = ig.Entity.extend({
  size: {x:20,y:40},
  offset:{x:6,y:4},
  vel: {x:0,y:0},
  maxVel:{x:200,y:200},
  health: 400,
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
.defines(function(){
GameInfo = new function(){
  this.score = 0;
},
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Firing Up Your First Impact Game

Now that we have seen in the *Preface* why the Impact Engine is a good choice for game developers, it's time to get to work. In order to develop a game, you will first need to set up your work environment. Just like a painter who needs his brushes, canvas, and paint, you will need to set up your code editor, server, and browser. By the end of this chapter, you will have equipped yourself with all the tools you need to start your adventure and even run a game on your computer.

In this chapter, we will cover the following topics:

- Setting up your own local server by using XAMPP
- Running a premade game on this server
- A short list of script editors you can choose from
- Debugging your game using browsers and the ImpactJS debugger scripts
- Some interesting tools you should consider for helping you create your game

Installing a XAMPP server

When developing anything at all, be it an application, a website, or a game, the creator always has a staging area. A staging area is like a laboratory; it is meant for building and testing everything before showing it to the world. Even after releasing a game, you will first make changes in the laboratory to see if things don't blow up in your face. Blowing up in *your own* face is fine, but you don't want this to happen to your gamers. Our staging area will be a local server, and during the course of this book we will use **XAMPP**. The X in XAMPP is an indication that this server is available for different operating systems (cross environmental, hence X).

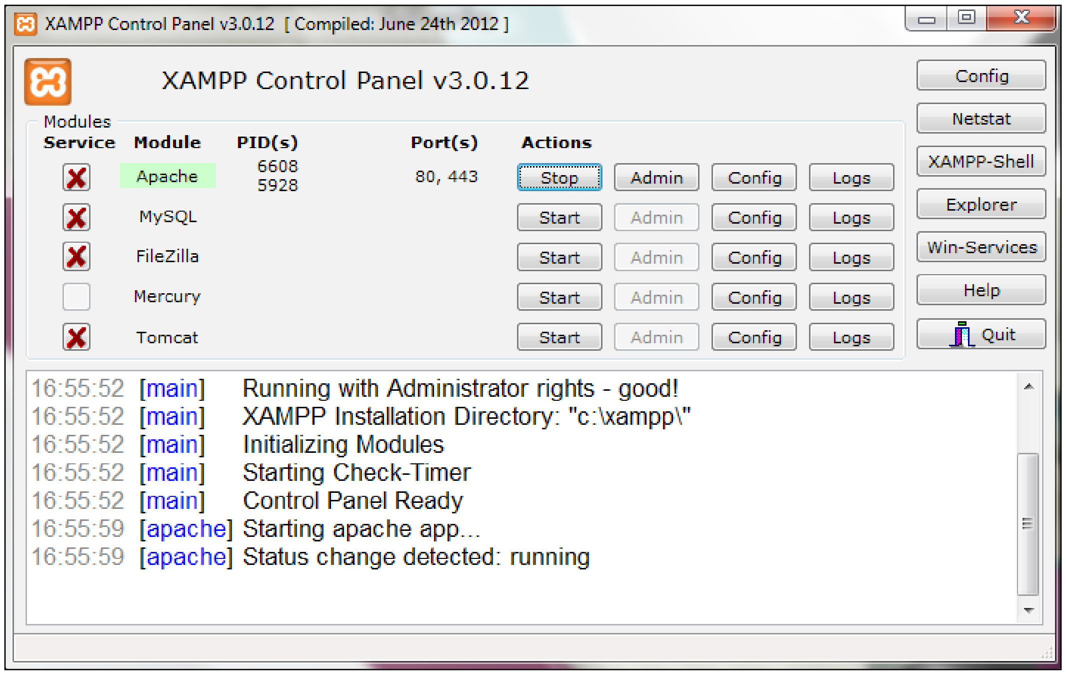
The other characters (**AMPP**) stand for **Apache**, **MySQL**, **PHP**, and **Perl**. You can install XAMPP on Windows, Linux, or Mac depending on which version you download and install. There are alternatives for XAMPP such as **WAMP** (for Windows) and **LAMP** (for Linux). These alternatives are fine too, of course.

Apache is the open source web server software that enables you to run your code. MySQL is an open source database system that enables you to store and query data with the SQL language. PHP is a language that is capable of connecting the SQL commands (which can manipulate a database) to the website or game code (JavaScript). Perl is often called "the Swiss army knife of programming languages" because of its versatility in uses. Installing a XAMPP server is rather straightforward.

You can go to the following website and download the appropriate installer for your system:

`http://www.apachefriends.org/en/xampp.html`

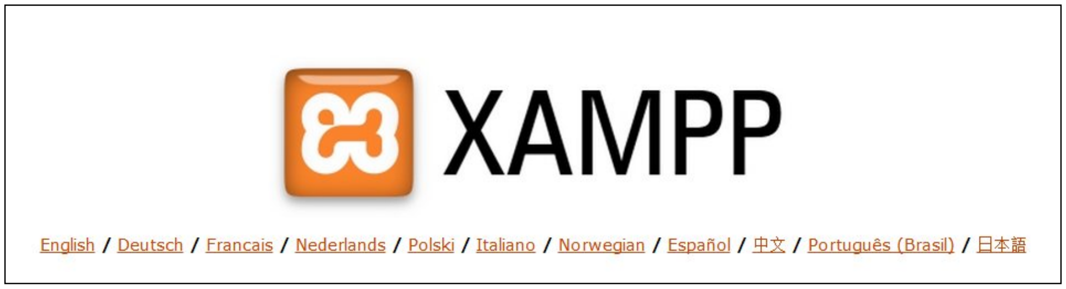
After installing the XAMPP server, which is basically going through a standard installation wizard, it is time to check out the **XAMPP Control Panel** page.



```
16:55:52  [main]    Running with Administrator rights - good!
16:55:52  [main]    XAMPP Installation Directory: "c:\xampp\"
16:55:52  [main]    Initializing Modules
16:55:52  [main]    Starting Check-Timer
16:55:52  [main]    Control Panel Ready
16:55:59  [apache]  Starting apache app...
16:55:59  [apache]  Status change detected: running
```

In this panel you can see the different components of the server, which can be switched on and off. You will want to make sure that at least the Apache component is up and running. The others can be on too, but Apache is absolutely necessary for running your game.

Now go to your browser. During the course of this book we will be using the Chrome and Firefox browsers. However, it is recommended to also install the latest Internet Explorer and Safari browsers for testing. In the address bar simply type `localhost`. Localhost is the default location for a locally installed server. Do you see the following *XAMPP splash screen*?


XAMPP
English / Deutsch / Francais / Nederlands / Polski / Italiano / Norwegian / Español / 中文 / Português (Brasil) / 日本語

Well congratulations, you have successfully set up your own local server!

A known issue is the occupation of your necessary ports by **IIS** (**Internet Information Services**). You might have to disable or even remove them in order to free your ports for XAMPP.

For **MAMP** (**M** stands for **Mac**), it is possible that you will need to specify the port 8888 in order for it to work. So type `localhost: 8888` instead of just `localhost`.

Summing up the preceding content, the steps will be as follows:

1. Download and install XAMMP.
2. Open the control panel and start Apache.
3. Open your browser and type `localhost` in the address bar.

Installing the game engine: ImpactJS

The next thing you will need is the actual ImpactJS game engine, which you can buy from the ImpactJS website at `http://impactjs.com/` or which comes as a package with **AppMobi** on the AppMobi website at `http://www.appmobi.com`.

Regardless of where you buy the engine, you will be looking for a (zipped) folder filled with JavaScript files. This is essentially what ImpactJS is, a JavaScript library that makes it easier to build 2D games in an HTML environment.

Now that you already have your server up and running and you have acquired this ImpactJS engine, all you need to do is put it in the correct place and test whether it works.

In the ImpactJS version (v1.21), at the time of writing this book you get one folder called `impact` and one `license.txt` file.

The license file tells you what you can and cannot do with your purchased Impact license, so you would be advised to at least read it.

The `impact` folder itself consists of more than just the Impact game engine; it also includes the level editor. The folder structure should be able to hold all your future game files.

For now, it is enough to know that you can copy the entire `impact` folder to the root location of your server.

For XAMPP this should be: `"your installation location"\xampp\htdocs`.

For WAMP this is: `"your installation location"\wamp\www`.

Let's also make a copy of this folder and rename it to `myfirstawesomegame` to make it a little more personal. Now you have the original folder that we will use later in *Chapter 3*, *Let's build a Role Playing Game* and *Chapter 4*, *Let's build a Side Scroller Game* next to the new folder called `myfirstawesomegame`.

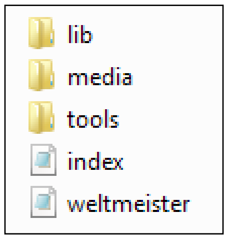
You should also have the following folder structure in both your XAMPP installation location as `\xampp\htdocs\myfirstawesomegame` and your WAMP installation location as `\wamp\www\myfirstawesomegame`.

In the `myfirstawesomegame` folder there should be the `lib`, `media`, and `tools` subfolders and the `index.html` and `Weltmeister.html` files.

Time for a little test! Just go to your browser and in the address bar type `localhost/myfirstawesomegame`.

The **it works!** message should now fill your heart with joy! If it doesn't appear on the screen, something went horribly wrong. If you don't get the message, make sure all your files are present and are in the correct location.

ImpactJS comes with a *physics engine* called `Box2D`. Check if you have this folder somewhere in your folder structure. If not, you can download a demo game that has the engine included via the personal download link that you got for downloading the Impact engine. This demo is a game called *Biolab Disaster* and here you should be able to find the `box2d` folder. If not, Dominic (the creator of ImpactJS) also made a separate folder called `physics` available. Since Box2D is a plugin for the standard engine, it is best to search the `plugins` folder in your `lib` folder as shown in the following screenshot and drop the `box2d` folder in here.



Summing up the preceding content, the steps are as follows:

- Buy the ImpactJS license and download its core scripts
- Put all the necessary files in a newly created folder called `myfirstawesomegame` in your server directory
- Open your browser and type `localhost/myfirstawesomegame` in the address bar
- Download the Box2D plugin and add it to the `plugins` folder on your own server

Choosing a script editor

We now have a server up and running with the ImpactJS game engine installed but we don't have a tool yet with which we can actually write the game code. This is where the *script editor* comes in.

For choosing the right code editor to suit your needs, it is best to make the distinction between a pure editor and an IDE. The **IDE** or **Integrated Development Environment** is both a script editor and a compiler. So this means that in one program you can both change and run your game. A script editor on the other hand is just for changing the script. It won't show you the output but will, in most cases, tell you in advance when you are about to make a syntax error. While the editor will show you syntax errors in your JavaScript code, actually executing the code will reveal logical mistakes and give you something (pretty) to look at.

For ImpactJS there is an IDE called AppMobi, which is free but charges for extra services. The alternative to using AppMobi is the XAMPP server you just installed.

Script editors, even very good ones, often come for free. Some nice ones you should check out before choosing the one you like are **Eclipse**, **notepad++**, **komodo edit**, and **sublime edit 2**. For Mac specifically there is **Textmate**, which is an often used editor but it doesn't come for free. And of course **Xcode**, the official apple developer editor.

All of these script editors will check for mistakes you make in the JavaScript code, however, they don't check for ImpactJS specific code. For this you could make your own script color coding bundle or download one from those who took the time to build one.

Download and install a few of the previously mentioned script editors and pick the one you like best. All of them will do the trick just fine, it's just a matter of preference.

Running the premade game

It's time to get a game up and running on your PC. In order to do this you will need the files that come with the book. These can be downloaded from the following site:

`http://www.PacktPub.com/support`

Now you should have everything. Copy the downloadable material for *Chapter 1, Firing Up Your First Impact Game*. Replace the `index.html` and `main.js` scripts that come with the ImpactJS library with the `index.html` and `main.js` scripts, which you can download from the Packt Publishing's download page. Also overwrite the `media`, `entities`, and `levels` folders on your PC with the ones provided.

Return to your browser and reload the `localhost/myfirstawesomegame` link. Lo and behold! A *fully functioning ImpactJS game*! In case you still see the **it works!** message as shown in the following screenshot, you might have to clear your browser cache or refresh the page a couple of times before the game shows up. In case something else is wrong, we will find out when we learn about debugging.

Summing up the preceding content, the steps are as follows:

- Download the necessary files from the packtpub download server and put them in the correct location on your own server
- Open your browser and type `localhost/myfirstawesomegame` in the address bar

Debugging your game with the browser and ImpactJS

Before you can debug the game, the least you should know is the *general structure of ImpactJS code*. As you might have noticed, ImpactJS has a main script that is used for actually steering the entire game. The `main.js` script includes all other necessary scripts and the `ImpactJS` library. Every script it includes represents a module. Like this, you have a module for every level and entity in the game. They are like Lego blocks, brought together in one big (`main.js`) castle. Indeed, the main script, as shown in the following code snippet, is a module in itself, requiring all other modules:

```
ig.module(
  'game.main'
)
.requires(
  'impact.game',
  'impact.font',
  'game.entities.player',
  'game.entities.enemy',
  'game.levels.main',
  ...
```

If you take a look inside a level script you will find it to be written in **JSON** (**JavaScript Object Notation**), which is a subset of **object literal notation**. JSON and normal literal differ from each other in the following ways:

1. The JSON keys are strings encapsulated in double quotes.
2. JSON is used for data transfer.
3. You cannot assign a function as a value to a JSON object.

More in-depth information about JSON can be found on `http://json.org/`. Literal is used throughout ImpactJS and looks like the following code snippet:

```
EntityPlayer = ig.Entity.extend({
  size: {x:20,y:40},
  offset:{x:6,y:4},
  vel: {x:0,y:0},
  maxVel:{x:200,y:200},
  health: 400,
```

Properties are defined with the colon (`:`) and separated by a comma (`,`). In normal JavaScript, this is done differently as follows:

```
if(ig.input.state('up') || ig.input.pressed('tbUp')){
  this.vel.y = -100;
  this.currentAnim = this.anims.up;
}
```

The equal sign (=) is used to assign a value to a property and the semicolon (`;`) is used to separate different properties and other variables.

Summing up the preceding content, the conclusions are as follows:

- Three types of JavaScript notations are used by ImpactJS: JSON, literal, and normal notation
- ImpactJS level scripts are in JSON code
- ImpactJS makes use of both literal and normal JavaScript notations

Debugging with the browser

While writing code with your freshly installed script editor, you will notice that JavaScript syntax mistakes can be avoided immediately since the editor will tell you what is wrong. However, some mistakes can only be found when actually running the code in the browser. Maybe you didn't want that princess to go up in flames when the hero saves her, but unintended results do happen. This means you should at all times have your browser open so that you can reload your game over and over until everything is to your liking.

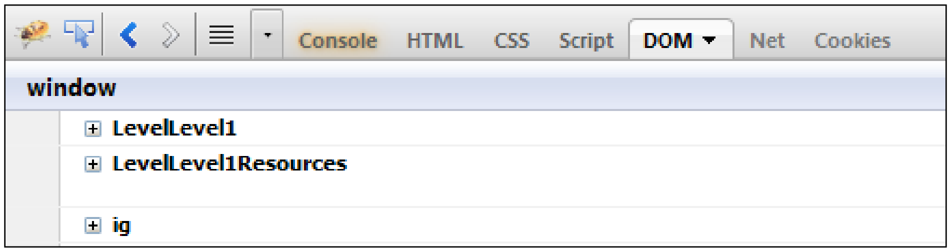
However, when your game crashes or doesn't even load completely, it can be a real pain to find the cause. Even when changing small chunks of code at a time, errors can pop up at unexpected places.

For this purpose both Firefox and Chrome have good tools at their disposal.

Firebug – the Firefox debugger

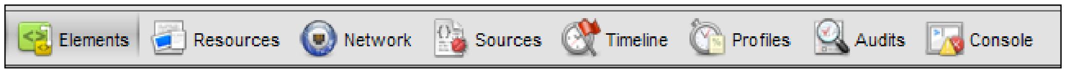
For Firefox you can install the **Firebug** plugin, a decent JavaScript debugger that will tell you on which lines of code your mistakes are, and has an easy explorer for the **DOM** (**Document Object Model**) of your game. The DOM is basically the structure of your HTML document containing all the entities and functions; having a good insight into the DOM is a must.

This DOM explorer is extremely useful for checking values of certain variables such as the health of your hero or the number of enemies on the screen. Mistakes that cause your game to crash will be easy to find with a debugger (both Firefox and Chrome). But in order to discover that you've added two extra zeros at the end of your enemy's health (so those creatures just won't die!), you will need to explore the DOM.



Chrome debugger

For Chrome you don't even have to download a plugin because it comes bundled with its JavaScript console. This console can be found under **Options** | **Extra** | **JavaScript console** and is a dream to work with. You can also bring up the console by right-clicking on your webpage and selecting **Inspect element**.



The Chrome debugger (also called Chrome developer tools) has eight tabs of which four are particularly useful for debugging Impact games.

The **Elements** tab allows you to inspect the HTML code of your page and even edit it on the spot. This way you can, for instance, change the canvas size for your game. You have to be aware though that changes are made only to the loaded web page; they are not saved to your HTML or JavaScript files.

In the **Resources** tab you can look up information about your *local storage*. Local storage is not necessary for building a game but it is a cool feature used for saving high scores and the like.

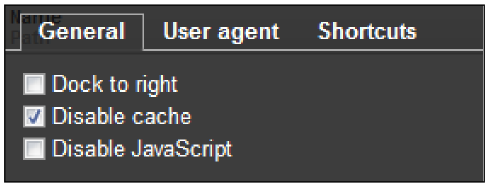
The **Sources** tab is extremely useful because it allows you to inspect and change (again temporarily) your JavaScript code. You can find your DOM in this tab like you would in Firefox. Code can be paused manually or by the use of conditional breakpoints. If, for instance, your character can gain experience, you can pause the game when leveling up to see if all variables take the value you expect them to.

The most important part of the debugger is the **Console** tab. The console shows where your errors are and even indicates the JavaScript file and line on which they occur. The console is pretty versatile such that it can be called while any other tab is open. This way you can check your code in the **Sources** tab and if you have errors, call the console by clicking on the **X** icon in the bottom-right corner. With both the **Sources** and **Console** tab opened, debugging becomes a walk in the park.

The other four tabs are the **Network**, **Timeline**, **Profiles**, and **Audits** tabs. They are useful but you will be spending most of your time with both the **Sources** and **Console** components opened.

During the course of this book, Firebug and Chrome debugger can be used interchangeably.

Changing your game and reloading your web browser is often not enough if your caching is enabled. As long as your game is cached, you are not 100 percent sure whether you are evaluating the latest version of your game or a previous one that the browser had locked in the memory. When developing games, it is wise to turn caching off. In Firefox it can be done by downloading and using a plugin that does just this. In Chrome it is simply an option of the Chrome developer tools itself. When the debugger is open, click on the cog-wheel symbol in the bottom-right corner to open the settings. Under the **General** tab you can disable the cache as shown in the following screenshot:



Dock to right [✓ Disable cache] Disable JavaScript

Debugging can be done in a single browser but it is wise to test whether your game runs smoothly at least on four browsers such as Chrome, Safari, Internet Explorer, and Firefox. With these four you cover at least 95 percent of browser usage.

Testing for certain devices will also be necessary if you want to launch a game for them. This can be done by owning one of the devices on which you want your game to work (for example, an iPad, iPhone, HTC, Galaxy, and so on) and placing your game online with the help of a web hosting company such as `one.com`. Alternatively, you can use AppMobi, which features a device viewer for this purpose.

Another great way to test your game is by using emulators. Emulators are programs that will simulate being an actual smart phone. This is all well and good but let's look at a practical example.

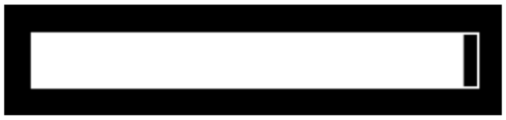
Exercises in debugging with Chrome and Firebug

In the previous sections we got the game up and running. Now let's see what happens if something is actually wrong (provided everything worked fine until now).

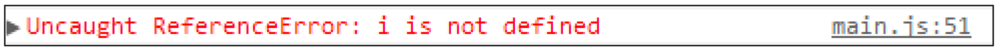
We need some *flawed code files* first. So copy the `main.js`, `player.js`, `projectile.js`, and `enemy.js` scripts from the `debugging tutorial` folder and replace the old ones with these scripts. `main.js` should be located in your `game` folder, and `enemy.js` can be found in the `entities` folder.

Now that your special (read: faulty) scripts are in place, it's time to restart the game. Reload your browser and make sure the cache is empty, otherwise no mistake will be shown.

The game does not load completely as you can see from the following loading bar:



This might happen to you regularly while working on a new game. This will, for instance, always occur if the dependencies of the different JavaScript files are wrong. To see what happened now, open the Chrome debugger.



```
▶Uncaught ReferenceError: i is not defined                    main.js:51
```

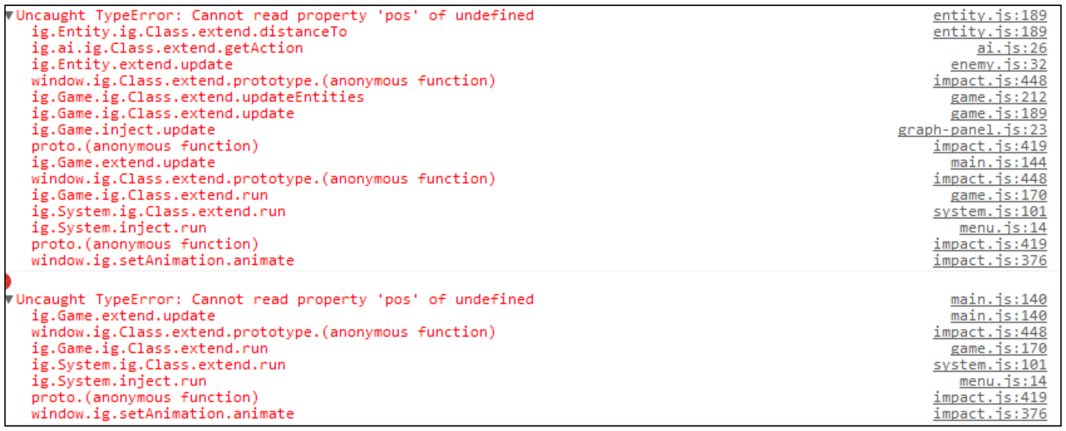
Go to the **Console** tab and check out the error message: **i is not defined main.js:51**. Open the `main.js` script in your editor at the indicated line number. And sure enough, there is something wrong as shown in the following code:

```
i.input.bind(ig.KEY.UP_ARROW, 'up');
ig.input.bind(ig.KEY.DOWN_ARROW,'down');
```

There is no object called `i`, this is supposed to be `ig` like the others.

Now that we fixed this issue, reload the game again. It loads! Great! This does not, by definition, mean everything is without errors. Open the debugger and see if anything else is bugging your game. Not at the moment, so let's get to exploring.

If all goes "well", your game should hang the moment you wanted to walk to the left.



```
▼Uncaught TypeError: Cannot read property 'pos' of undefined                entity.js:189
   ig.Entity.ig.Class.extend.distanceTo                                     entity.js:189
   ig.ai.ig.Class.extend.getAction                                              ai.js:26
   ig.Entity.extend.update                                                   enemy.js:32
   window.ig.Class.extend.prototype.(anonymous function)                   impact.js:448
   ig.Game.ig.Class.extend.updateEntities                                    game.js:212
   ig.Game.ig.Class.extend.update                                            game.js:189
   ig.Game.inject.update                                              graph-panel.js:23
   proto.(anonymous function)                                              impact.js:419
   ig.Game.extend.update                                                     main.js:144
   window.ig.Class.extend.prototype.(anonymous function)                   impact.js:448
   ig.Game.ig.Class.extend.run                                               game.js:170
   ig.System.ig.Class.extend.run                                           system.js:101
   ig.System.inject.run                                                      menu.js:14
   proto.(anonymous function)                                              impact.js:419
   window.ig.setAnimation.animate                                          impact.js:376

▼Uncaught TypeError: Cannot read property 'pos' of undefined                  main.js:140
   ig.Game.extend.update                                                     main.js:140
   window.ig.Class.extend.prototype.(anonymous function)                   impact.js:448
   ig.Game.ig.Class.extend.run                                               game.js:170
   ig.System.ig.Class.extend.run                                           system.js:101
   ig.System.inject.run                                                      menu.js:14
   proto.(anonymous function)                                              impact.js:419
   window.ig.setAnimation.animate                                          impact.js:376
```

You get the message, **Cannot read property 'pos' of undefined**. The problem is it's so hard to locate where the error actually occurs because an error seems to show up in almost every script. However, what we do know is that `pos` is a parameter of an entity and that we have pressed the `left` button before the error occurred. We should at least check all places where this `left` button is defined or used.

If you open the `player.js` script, you will find that the commands for moving left are rather peculiar as shown in the following code:

```
else if(ig.input.state('left') || ig.input.pressed('tbLeft')){
  this.vel.x = -100;
  this.currentAnim = this.anims.left;
  this.kill();
}
```

So the entity moves to the left, has its animation set to left, and then kills itself. Throwing around the random `kill()` functions is not a good thing. In this case, the unexpected spot for the `kill()` function caused the player to disappear and thus the player has no more position, which created errors further down the `update` loop of the game. Remove this `kill()` function and no more game crashes should occur.

So sometimes the console shows an error, but it's still your wits that will lead you to the root cause. The console is only a tool, you are the real mastermind.

We have removed all major bugs because Chrome does not indicate bugs at the current moment. Make sure to check all levels since different levels often have other entities that might be buggy. However, for now, let's get to killing some enemies!

You might have noticed that it's considerably hard to destroy one of those nasty creatures. We don't have any real errors anymore, but maybe something else is not going according to the plan. We can't seem to kill it, so either we don't cause enough damage or it has an enormous amount of health. Let's dive into the two entities that might be involved: `projectile` and `enemy`. We should check the `projectile` entity and not the `player` entity because even though the player shoots the projectile, it's the projectile that does the damage. Guns don't kill people, bullets do. Open the `projectile.js` and `enemy.js` scripts, which are both in the `entities` folder. Alternatively, you can open the Chrome debugger and select the files under the **Scripts** tab.

In the `projectile.js` script, look for the following code:

```
check: function(other){
  if(other.name == 'enemy'){other.receiveDamage(100,this);}
  this.kill();
  this.parent();
```

We will dive into the specifics of this code soon. For now it is enough to know that a bullet does `100` damages upon impact with an enemy.

Now check out the health of the enemy in the `enemy.js` script. The following code shows the health:

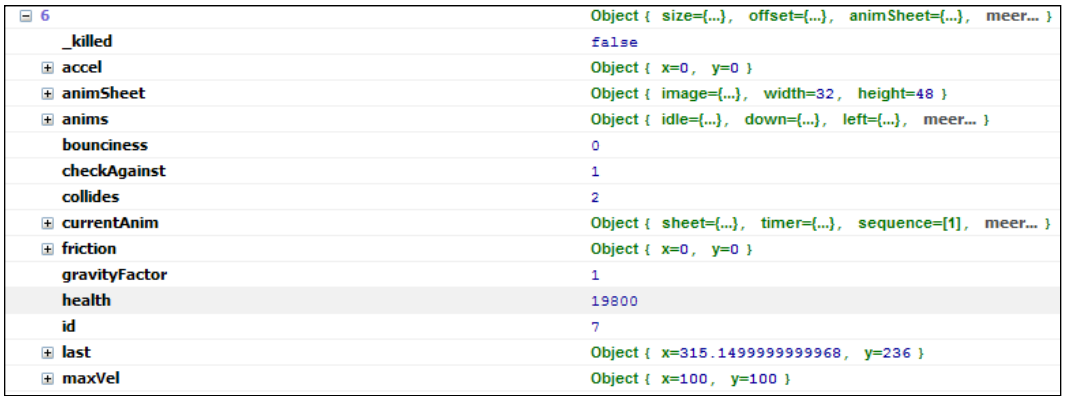
```
health:200000,
```

Yes. That's a problem. The enemy is a thousand times stronger than intended. Change the health to `200` and you will be able to kill your enemy with two shots. Alternatively, you can set the damage of the `projectile` entity to `100,000`. Changing the `damage` property to a large number can be useful in order to appeal to players who prefer to see big numbers over modest ones. (those who ever played World of Warcraft know what I'm talking about).

If you save the code and reload the level, you should have no more problems killing your enemies.

An alternate way of finding out what might be wrong is by taking a look at the individual entity by exploring the DOM. Let's use Firebug for this. If you don't have it installed on your Firefox just yet, Google it and install it.

We shot the enemy twice and found out he has no intention of dying. Instead of checking the code, we can have a look at the spawned entity itself by navigating through the DOM. To find the enemy's health you must open your Firebug by pressing the bug symbol in your browser and then select the **DOM** tab. Now open the `ig`, `game`, and `entities` folders in that order. You will see a numbered list, the number being the position of a particular entity in the `entities` array. You can now open some of the numbers until you find the enemy with **19800** as the **health** value, as shown in the following screenshot:



```
⊟ 6                Object { size={...}, offset={...}, animSheet={...}, meer... }
    _killed        false
  ⊞ accel          Object { x=0, y=0 }
  ⊞ animSheet      Object { image={...}, width=32, height=48 }
  ⊞ anims          Object { idle={...}, down={...}, left={...}, meer... }
    bounciness     0
    checkAgainst   1
    collides       2
  ⊞ currentAnim    Object { sheet={...}, timer={...}, sequence=[1], meer... }
  ⊞ friction       Object { x=0, y=0 }
    gravityFactor  1
    health         19800
    id             7
  ⊞ last           Object { x=315.1499999999968, y=236 }
  ⊞ maxVel         Object { x=100, y=100 }
```

The enemy is buried in a list of other entities, but by his properties we can see what is going on here. We shot it twice and now it has a **health** value of **19800**. That makes sense because the damage of the projectile is **100**.

Mastering the DOM takes a bit of effort but it is very useful to find out if your code works as intended. In addition, you get a better insight on how the core elements of ImpactJS are related to each other. It's recommended to spend some time here to get a feel of the overall structure before proceeding.

So we have seen three different types of errors, which rank from easily solvable to rather difficult to find and fix. In the first scenario, the console tells you that there is an error at one spot and you go and set it right. The second scenario shows the game producing errors on several spots at once with a single root cause. It is up to you and your logical brain to deduct what crashed the game. Finally, we have the unexpected results, which are not really errors. The console doesn't show an error for these because it can't read your mind (maybe in a next version, who knows). These are hardest to find and will take some testing from your side.

Summing up the preceding content, the conclusions are as follows:

- Both Firefox and Chrome features are very capable debuggers.
- Firebug is especially recommended for exploring the DOM of a game.
- Chrome has eight interesting components of which the most useful one is the console, which detects errors.
- Errors can come in different types: syntax errors, code logic errors, and game logic errors.
- Most syntax errors can be detected preemptively by a good script editor.
- A simple syntax error will show up in the debugger console as a single line error. This makes it easy to locate and repair.
- Code logic errors can be hard to detect since the syntax is often correct at its root but will show errors on other locations.
- Game logic errors are very subjective errors since they will not make the game crash but result in poor gameplay.

Debugging with ImpactJS

ImpactJS itself comes with a debugger built into the engine. However, it is turned off by default and can be turned on by making a small modification to the `main.js` script. The `main.js` script is (as the name suggests) the main script of your game and calls all your other JavaScript files. It is this script that is loaded into the HTML canvas in your browser and is looped through over and over again in order to make your game run. The `main.js` script can be found in the `game` folder and should have come together with your Impact license, as shown in the following code snippet:

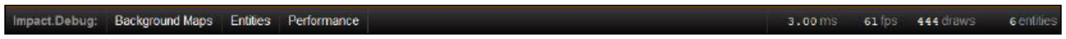
```
ig.module(
  'game.main'
)
.requires(
  'impact.game',
  'impact.font',
  'impact.debug.debug',
```

Everything starts with the `ig` (Impact Game) object. This object is what you will be looking for in your DOM when you are debugging your game and checking variables and functions. In the `main.js` script, there is a call to the `.module` function, which defines `game.main` to be a module of your game. The module name needs to be the same as its location and filename! So the JavaScript file `lib/game/entities/bigmonster.js` will be finally defined as `game.entities.bigmonster`. Adding the `debug` panel to the game can be done by following these steps:

1. The `.requires()` function calls all the scripts it needs to be able to execute the code successfully. Not every module will need this but the `main.js` script will always at least require the `impact` library.
2. It is in this function call that you will want to add the `impact.debug.debug` script, which (as you guessed) calls the script `debug.js` in the folder `lib/impact/debug`.
3. Save the `main.js` script and rerun `localhost/myfirstawesomegame` in Chrome.
4. If everything goes according to the plan, you should now see a new toolbar at the bottom of your browser called **impact.debug**.
5. The debugger has three tabs: **Background Maps**, **Entities**, and **Performance** and four key indicators in the top-right corner.
6. These indicators from left to right are as follows:
    - The number of milliseconds needed to run one frame of the game.
    - The **fps** indicator or frames per second of the game.

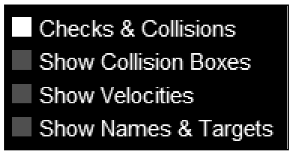
  - The number of **draws** that have taken place. This includes characters if you have a conversation going on.
  - On the right-hand side you find the number of entities currently in the game.

While these indicators quickly show you the most important things to take into account, the three tabs as shown in the following screenshot go somewhat more in depth:



If you select **Background Maps**, you will see all the graphical layers the game has. Let's say you want to run in front of a tree with your epic character; you will expect part of the tree to disappear behind the character and not vice versa. When the character moves behind the tree you will want it to be hidden by it. So you see, you need at least two layers in order to draw a tree like this. One layer comes in front of the player (most likely the treetop), the other behind it (the trunk).

In the part of the debugger, as shown in the following screenshot, you can turn layers on and off. If a layer is set to be prerendered, you will be able to see the chunks of the layer. In the following screenshot, **Checks & Collisions** is turned on while the other options are turned off:



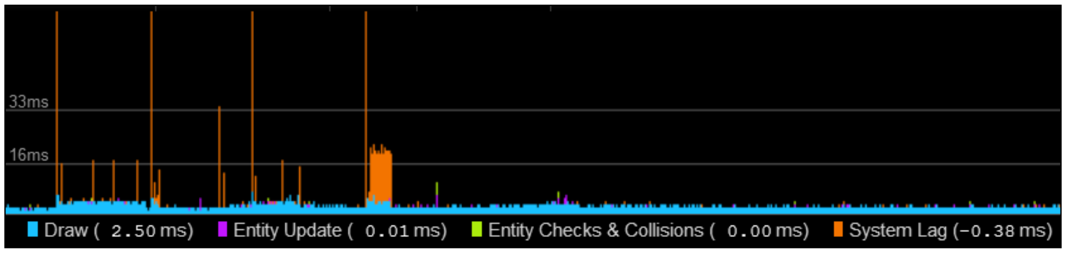
- ☑ Checks & Collisions
- ☐ Show Collision Boxes
- ☐ Show Velocities
- ☐ Show Names & Targets

With the **Entities** tab you can turn several interesting indicators on and off. If you turn on **Show Collision Boxes**, you will be able to see a red box around your character and several (invisible) entities that constantly check for collision. These red boxes indicate the borders at which the point collision is triggered. This is important because if the collision box around your hero character is much bigger than the image, he might not fit through doors anymore or get mysteriously hit by enemies who are still far away. You can set the size of these collision boxes yourself when writing your code, allowing for some interesting effects, such as only being able to kill a boss by shooting him in the eyeball.

When you turn on **Show velocities**, you should walk around with your character. You will now see a line sticking out in front of him, which is an indication of how fast he is currently going.

By showing names and targets you can see all named entities and their targets. It's a fun feature but you are better off with the ImpactJS level editor (**Weltmeister**) for your purpose.

Finally the **Performance** tab shows you how much effort goes in the different tasks that the browser needs to perform for running your game, as shown in the following screenshot:



Two horizontal lines are visible on the previous chart: the **33ms** and **16ms** lines. These lines correspond to an approximate frame rate of 60 fps and 30 fps. Having a game run below 30 fps is inadvisable since it will look as if the game is lagging, and there is no fun in playing such games. If you see that your game lags, check which part is taking all the juice and try to fix that.

In most cases, drawing the game (graphics) takes up most of the resources. This is indicated by **Draw** in the **Performance** tab. If this happens, try to work with fewer layers or bigger tiles. Also prerendering can increase performance in this case.

Another part of the resources is taken up by your entities and how they interact. If you have thousands of different entities swarming over your screen because you decided that a flock of seagulls should be represented by a separate entity for each bird, you might get into trouble quickly.

There is a separate indicator for system lag, which is a parameter you have no power over since this shows how well the browser performs. Quite regularly the system lag creates spikes that drop the frame rate. However, in most cases it cannot really be felt because the really huge spikes come and go so quickly.

Summing up the preceding content, the conclusions are as follows:

- ImpactJS comes with its own debugger but is turned off by default
- The debugger has several components that allow insight into entity behavior, collision, and game performance
- ImpactJS debugger is extremely useful for tracking performance issues and should always remain turned on during development

Which helpful tools are out there

If you have a decent script editor, the ImpactJS library, a (local) server, and a browser with debug capabilities, you are set to build an ImpactJS game. However, there are several interesting tools out there that can facilitate your journey quite a bit. There is **Ejecta** that comes with ImpactJS and is a more efficient way of releasing games for iPhone and iPad. **AppMobi** is a great tool for preparing your game for the webstores. **PhoneGap** is an open source framework for creating apps. With **lawnchair** you can more easily make use of local data storage. There is **Scoreoid**, a free game cloud service. And finally **Playtomic**—a game analytics tool. In this chapter, we will take a short peak at each one of these.

Ejecta

Ejecta is a fine piece of ingenuity, which is free to download at the following link:

`http://impactjs.com/ejecta`

It totally replaces **iOSImpact**, which was a way of preparing the game as a native app for the Apple store. Dominic calls Ejecta a "browser without the browser". It has no overhead, only the canvas in which your game features and the audio elements.

Ejecta works well for ImpactJS but it can be used for other apps as well. Like the former iOSImpact, it makes use of **OpenGL** for the animation and **OpenAL** for the audio, which drastically increases game performance. If you plan on releasing your game to the iPhone, Ejecta is certainly worth taking a look at.

AppMobi

AppMobi offers an **XDK** (**Cross platform Development Kit**), which goes very well with ImpactJS. So well that they actually have a separate development kit for ImpactJS (Impact XDK) apart from their normal one (AppMobi XDK).

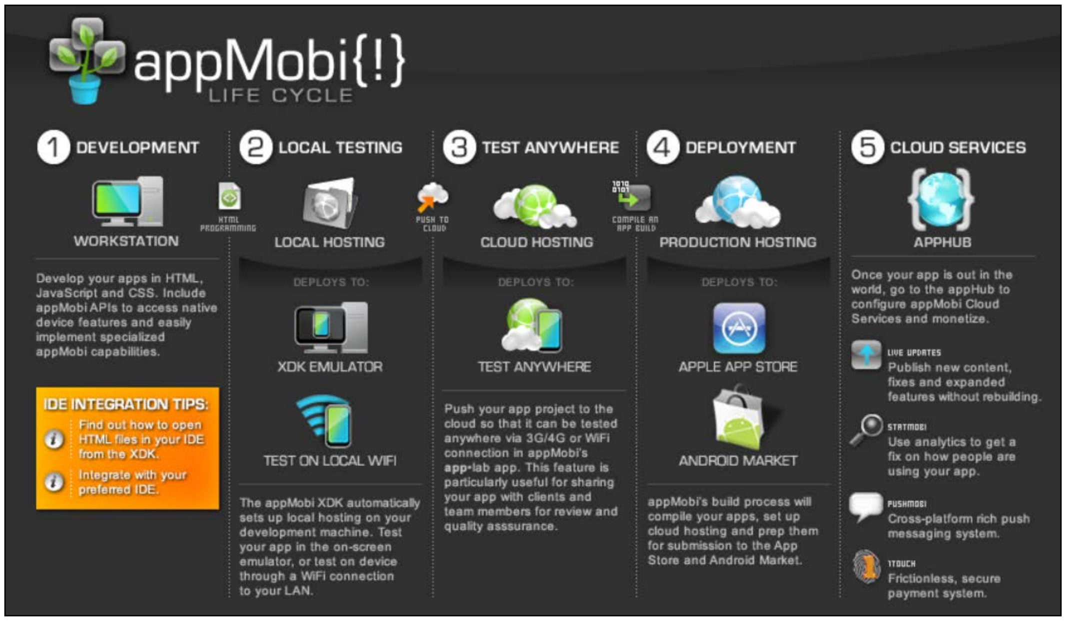
Use of the development kit is free but extra services such as their cloud service, live update features, and secure payment are available for an extra fee. You can find everything on `http://www.appmobi.com/`.

The Impact XDK will only allow you to work on an Impact Game if you have registered your Impact key in your account and included their JavaScript library. When set up properly, the XDK allows you to simulate several devices such as iPad, iPhone, Galaxy, and so on. The XDK only runs in Google Chrome, though that isn't really much of a weak spot. You can open a script editor, but it is in no way as good as the ones we looked at before. You have the option of calling a debugger but it is just the Google Chrome debugger, not one they built themselves.

The **Apphub** (which is your control center) allows you to build and test apps before sending them to the shops. In order to release a game you will, of course, still need the developer accounts for the platform you want to service.

AppMobi also has what they call **direct canvas acceleration**, which increases the game's performance by bypassing the canvas element of your games. It is very similar to what Ejecta does but then it is delivered by AppMobi.

The following screenshot is an overview of the different terrains that AppMobi can provide, which will give relief to some developers. While AppMobi is only of limited use while scripting your game, it can provide excellent support during testing and deployment.



Develop your apps in HTML, JavaScript and CSS. Include appMobi APIs to access native device features and easily implement specialized appMobi capabilities.

IDE INTEGRATION TIPS:

Find out how to open HTML files in your IDE from the XDK.

Integrate with your preferred IDE.

The appMobi XDK automatically sets up local hosting on your development machine. Test your app in the on-screen emulator, or test on device through a WiFi connection to your LAN.

Push your app project to the cloud so that it can be tested anywhere via 3G/4G or WiFi connection in appMobi's **app**•lab app. This feature is particularly useful for sharing your app with clients and team members for review and quality asssurance.

appMobi's build process will compile your apps, set up cloud hosting and prep them for submission to the App Store and Android Market.

Once your app is out in the world, go to the appHub to configure appMobi Cloud Services and monetize.

LIVE UPDATES
Publish new content, fixes and expanded features without rebuilding.

STATMOBI
Use analytics to get a fix on how people are using your app.

PUSHMOBI
Cross-platform rich push messaging system.

1TOUCH
Frictionless, secure payment system.

There is no possibility of directly connecting to a mobile device. However, you can send a link to anyone who owns one. This way your friends can test your latest creation if they have the **AppMobi applab** installed.

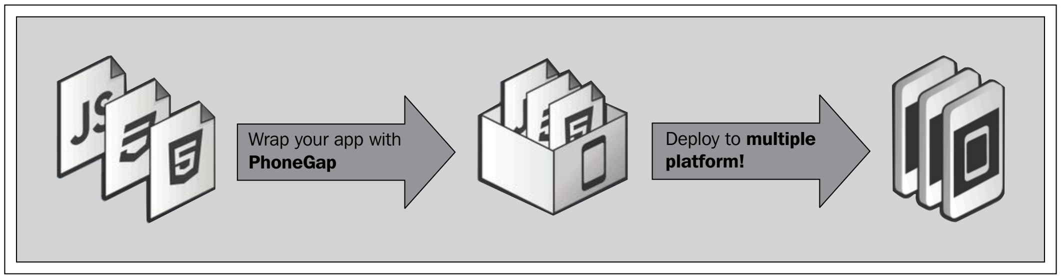
All in all, AppMobi is easy to get started with and certainly worth considering if you want some help with the entire process of releasing a game, though for development you are almost entirely on your own.

PhoneGap

PhoneGap (formerly known as **Cordova**) is another XDK for developing mobile-native applications.

PhoneGap can be compared to AppMobi with respect to functionalities but AppMobi is very visual and more newbie-friendly. PhoneGap enables you to build apps that are native to the **OS** (**operating system**), integrate **PayPal**, and use push notifications.

As shown in the following screenshot, PhoneGap provides a way to build your app in order to distribute to the different channels:


Wrap your app with **PhoneGap**
Deploy to **multiple platform!**

Getting started with PhoneGap is a bit more complicated than AppMobi. You will need to install **eclipse** (which is free), the android development tools, and SDK. Installing **Git** might be necessary for targeting specific platforms. If you want to release for iPhone or iPad, you will also need **xcode**.

In all, it's definitely worth taking a look at. Luckily they have very good documentation because it tends to get a bit complicated. More information can be found on `http://phonegap.com/`.

lawnchair

lawnchair offers a free and easy way of using **local storage**. Local storage is used for storing your data (saving files and high scores) on devices that run your game.

There are many advantages in saving everything on the client side over saving on the server side. For one, you don't need to know SQL. Websites save everything in their databases by the use of SQL, PHP, and JavaScript. If you use local storage, you only need JavaScript. The amount of storage space is not restricted by your server but by your user. So if you have many gamers who each use a small amount of space, you will never get into trouble with local storage while you might get into trouble when solely using server storage. Since it doesn't need to be transmitted to a server at all times, you can play your games offline and still keep your saves.

Those are all pretty good advantages but how does lawnchair work? lawnchair is a JavaScript library just like ImpactJS (but a free one this time). You only need to include it with your other JavaScript files and you can start using the specific commands to save data.

Including the lawnchair functionalities can be done by downloading the library from `http://brian.io/lawnchair/` and including the `lawnchair.js` script in your `index.html` file as shown in the following code example:

```
<html>
  <head>
    <title>my osim app</theitle>
  </head>
  <body>
    <script src="lawnchair.js"></script>
    <script src="app.js"></script>
  </body>
</html>
```

lawnchair uses JSON to save data in the DOM of your game. If you want an example of what this looks like, you can open any ImpactJS `level` script in the code editor of your choice since these scripts are also coded with JSON.

You will want to take a look at lawnchair if your game is going to require **save games**, **high scores**, **game progression**, or any other option that needs to be kept track of so the player wouldn't need to start all over. More information can be found on `http://brian.io/lawnchair/`.

Scoreoid

Scoreoid is a **gaming cloud service** aimed at taking care of some advanced features such as **leaderboards**, **player login**, and **ingame notifications**.

In order to use Scoreoid and its functions, you need to register on their website and implement their code within yours where necessary. There are different codes for different functions. The following code snippet is an example template for storing information about people loading your game:

```
$.post("API URL",{api_key:"apikey",game_id:"gameid",response:"xml"},
  function(data){
    alert("Data Loaded: "+ data);
    console.log("Data Loaded: "+ data);
  });
```

You fill out the **API URL**, your own **API key**, **game ID**, and the type of **data coding** for transferring (XML or JSON) and you're good to go.

An account is free but they have the option of a premium account, which is also free. But that is just because they are currently still working on defining the extra features for a premium account. You can subscribe on their website `http://www.scoreoid.net/`.

Playtomic

Playtomic is the Google analytics of game monitoring.

The basic account is for free but the premium one is currently priced at *$15/month* or *$120/year*. You can subscribe on their website `http://www.playtomic.com`.

Getting the flow of analytics up and running isn't too difficult. In your `index.html` file you include a reference to their JavaScript library as shown in the following script:

```
<script type="text/javascript"src="http://api.playtomic.com/js/
playtomic.v2.1.min.js">
</script>
```

In your `main.js` script, you can then add a command to send the data to their servers as follows:

```
Playtomic.Log.View(gameid, "guid", "apikey", document.location);
```

These two pieces of code are suggested by Playtomic. However, an error might occur if you send the data to their server in plain text. Therefore it is best to replace the script type `text` by `application` as shown in the following code snippet:

```
<script type="application/javascript"src="http://api.playtomic.com/js/
playtomic.v2.1.min.js">
</script>
```

one.com webhost

A **webhost** is what you will need if you want to place your own game on your own website.

You don't always need your own website since cloud hosts such as Scoreoid also allow you to put your game online. However, sometimes it's nice to have your own little place on the World Wide Web.

`one.com` sells web space and a domain name in one package. The price for this service is pretty ok, especially in comparison with what you need to do to get the same result. You would need to have a PC with XAMPP installed, and it should be up and running at all times. In addition, you would still have to buy a domain name if you are serious, or get a free domain from somewhere and reroute your IP to it. If your IP would remain the same at all times, this would be feasible. However, more often than not, this is a premium service of Internet providers. You can get an account at `http://www.one.com`.

If you want to use web hosting, more providers are out there, but downloading and installing **FileZilla** is recommended in all cases. FileZilla is an efficient file transfer program, which is just what you need to get all your files from your PC to a server somewhere in the desert. FileZilla can be downloaded at the following link:

`http://filezilla-project.org/`

Summing up the preceding content, the conclusions are as follows:

- There are a lot of tools out there that will make your life as a game developer much more agreeable
- Ejecta is an efficient solution for releasing games to iPad and iPhone
- AppMobi is a free cloud tool, which can help with releasing and developing almost every distribution channel
- PhoneGap has a lot in common with AppMobi albeit slightly more complicated to work with
- lawnchair provides a way of addressing local data storage
- Scoreoid is a free game cloud service, which will host your game and supply features such as leaderboard integration
- Playtomic is a game analytics tool allowing you to tag certain elements of your game and store the data on their servers

Summary

In this first chapter we have made our preparations as a game developer. We have set up a local server, which we can use as a development and initial test environment. In order to write our code we need a script editor, so we briefly went over some of the editors out there. Debugging is one of the main skills of a programmer; for this we can not only use the Chrome and Firefox debuggers but also the ImpactJS debug module. We finished by taking a look at several very helpful tools for ImpactJS game development.

Now we have everything ready to go. We will dive into ImpactJS by messing around with a small premade example game in the next chapter.

2
Introducing ImpactJS

Now that we have gathered all the necessary tools and got the first game up and running, it's time to learn more about how Impact actually works.

However, before diving into the code, we should first copy the code from the `chapter 2` folder to the correct locations.

Similar to what we have done in *Chapter 1*, *Firing Up Your First Impact Game*, we just need to overwrite the `main.js` and `index.html` files and the `entities`, `levels`, `plugins`, and `media` folders of the `myfirstawesomegame` project.

We are now set to go and explore the intricate workings of the ImpactJS engine!

In this chapter we will cover the following processes:

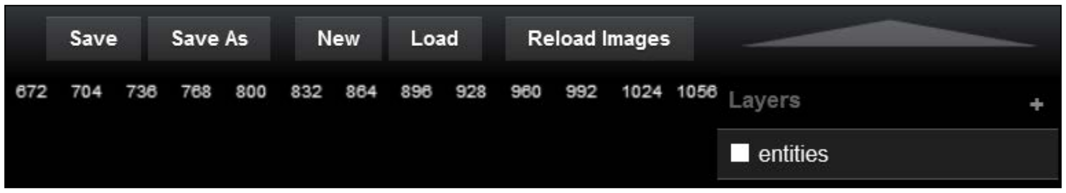
- The Weltmeister tool in ImpactJS and the consequences of changing certain parameters in a level
- The manner in which level layers impact level design
- How collision is handled in ImpactJS
- The ImpactJS entity
- The properties of an ImpactJS entity
- How a playable entity differs from a non-playable one
- How to spawn or kill a character
- How to set up player controls
- How to change the game's graphics
- How to play sound effects when triggered and a background tune
- How to add physics to a game using Box2D

Building your own levels

While designing a game, you would want to create the setting and the place where it all happens. Many games are divided into levels, often getting more difficult with each level upgradation. For some other games such as **RPGs** (**Role Playing Games**), there is no such thing as a level because it implies a certain cutoff with often no way back. Throughout the book, a space that can be saved as a single file in the Weltmeister will be referred to as a level.

The Weltmeister is literally the tool to master your world in ImpactJS. If you installed ImpactJS correctly you should be able to access the level editor shown in the next screenshot by placing the following address in your browser:

```
http://localhost/myfirstawesomegame/weltmeister.html
```



Creating, loading, and saving levels in Weltmeister

Creating levels for a game is one of the most enjoyable things to do in game design. The Weltmeister is so well put together that you will spend hours messing around with it just because you can.

On opening Weltmeister (by default), it starts with a clean slate; there is a vast amount of emptiness for you to fill. Soon we will get to building a level from scratch but for now we should load the `level1` level. Press the **Load** button in the top-right corner of your Weltmeister and select it in the `levels` folder. If you copied it in the beginning of the chapter it should be right there, otherwise copy it to the Weltmeister now.

`Level1` is a rather original name for a first level but let's personalize it a bit by saving it as `myfirstepiclevel`. Press the **Save As** button in the top-right corner and save it in the same directory. Now we have a copy to work on and mess around with.

Before we actually use the level called `myfirstepiclevel`, we will need to make a change in the `main.js` script's code:

1. Open the `main.js` script in your preferred script editor.
2. In the `main.js` script, you will see a call to the `loadLevel()` function.

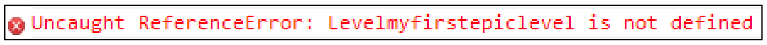
   ```
   this.loadLevel(LevelLevel1);
   ```

> This call is situated in the `init()` function of the game (`ig.game.init`). This means one of the first things the `main.js` script will do (at *initiation*, thus `init`) is load the level, `Level1`. Obviously, we don't want that anymore, as we have our own level called `myfirstepiclevel` now. In order to let the game know that it has to include this level, you will need to add it to the `.requires()` function as shown in the following code lines:
>
> ```
> 'game.levels.level1',
> 'game.levels.myfirstepiclevel',
> ```

3. Also, change the call to the `loadLevel()` function so it calls the level, `myfirstepiclevel`, instead of `Level1` as shown in the following snippet:

   ```
   this.loadLevel(LevelMyfirstepiclevel1);
   ```

> As you might have noticed, you always have to put the word `Level` before your actual level name. In addition you will always need to write both `Level` and your level name with a capital letter. Failure to comply with either of those will result in an epic crash at game load. Putting `Level` in front of the actual level name is a rather odd convention, especially since functions like `loadlevel()` are built to expect a level file. It is possible that this mandatory prefix is removed in the future versions of ImpactJS. But for now, failing to insert the word `Level` before your actual level name or writing both `Level` and your level name with a capital letter will result in the display of the following error:



```
Uncaught ReferenceError: Levelmyfirstepiclevel is not defined
```

The other buttons in the Weltmeister are **Save**, **New**, and **Reload Images**. The **Save** button just saves the file you are working on, and the **New** button will open a new and empty file. The **Reload Images** button is a refresh button for your tilesets. The tilesets of a game are collections of images. All the graphics of a single theme can be on a single tileset, for example, the `outdoor` tileset. Because several images are stored in a single overall image called a tileset, it is easier to create your level while working in the Weltmeister. You can look at it as the color palette of an artist, but as a level creator you have as many palettes at your disposal as you have tilesets.

Summing up everything we came across, we can conclude that:

- You can access the Weltmeister by typing the following address in your browser while your server is turned on: `localhost/myfirstawesomegame/weltmeister.html`
- Open `level1` by using the **Load** button
- Save it again as `myfirstepiclevel` with the **Save As** button
- Include the new level in the `main.js` script by adding `myfirstepiclevel` to the `include()` function

Layers and the z axis

With the level opened, you can see the different elements and layers of which it is comprised. Let's first have a look at the **Layers** menu on the right-hand side of the editor.

Select the **collision** layer and you will see the standard properties that need to be filled out for a layer in order to exist. All layers (except for the **entities** layer) have a name, tileset, tilesize, dimensions, and distance.

Tilesets are basically chains of square-formed images, which when put together well enough, form your idyllic looking landscape or scary dungeon. The tilesize is the width and height of one tile measured in pixels. Since all tiles are squares, you only need to fill out a number. The dimensions of a layer are the width and height of the entire map on which the layer needs to be present, measured in number of tiles. So a layer with a tilesize of 8, a width of 20, and a height of 30 is composed of 4800 (8 x 20 x 30) pixels. Take this into account while working with mobile devices. A level with a resolution of 160 (8 x 20) x 240 (8 x 30) will fit most devices. However, if the tilesize is 32 you will need a viewport that automatically follows your playable character in order to show off your level. This view-port is rather easy to incorporate and will be explained later in this chapter. To create a new tileset follow these steps:

1. Try creating a new layer by clicking on the plus (**+**) sign at the top of the layer selection menu.
2. Type in a name for the layer; let's say `astonishinglayer` or `tree`, whatever you like.
3. Now select the tree tileset from the `media` folder by clicking the empty box next to the **Tileset** field. If you can't reach it by using the Weltmeister menu, you just type `media/Tree.png` in the tileset box. Set the tilesize to `32` and the dimensions to `30 x 20` (width x height). You can see the border of your layer change accordingly.

It is a common mistake to have one layer smaller than the other and then not be able to add objects on one part of your map. So let's say you intend your level to be a map with dimensions of 30 x 20 and a tilesize of 32, and you add a layer like this, and fill it with grass. You want to add a bench on the grass so you add another layer and make the dimensions 30 x 20. Because your bench is a 32 x 16 image, you set the tilesize to 16. If you do this you will be able to draw your bench quite accurately but only in the top-left corner of your level. You will need to change the dimensions to 60 x 40 in order to occupy the same space as the grass layer does.

Distance is the speed at which the layer moves relative to your game's screen position. A value of `1` in the **Distance:** field means it moves at an equal speed, whereas the value `2` means the layer moves at half the speed. By setting this parameter higher than `1`, it becomes possible to make things appear further away; this is ideal for your nice cloudy background in a side scrolling (or parallax) game, such as Mario. Go to your game and walk your character from the far left edge of the game towards the right boundary to observe the effect of altering the value of the **Distance:** field.

Now return to the Weltmeister and try setting the value of the **Distance:** field to `2`. Save and reload the game, run your character from one edge to another edge of the level, and see what happens. Part of the game will appear to move slower than the rest. This is useful in side scroller games as a background, but it is also used in top-down games to create the impression of a terrifying abyss. Underneath, you have the options **Is Collision Layer**, **Pre-Render in Game**, **Repeat**, and **Link with Collision**. Turning them on and off can be done by clicking on the white squares (which turn black to indicate the option is switched off).

The **Is collision layer** option will tell the level editor that the objects in the layer you are drawing are impenetrable. Pre-rendering a layer will cause the game to cluster tiles while loading. This will increase the initial load time but decrease the number of draws the game requires and, thus, increase performance while running.

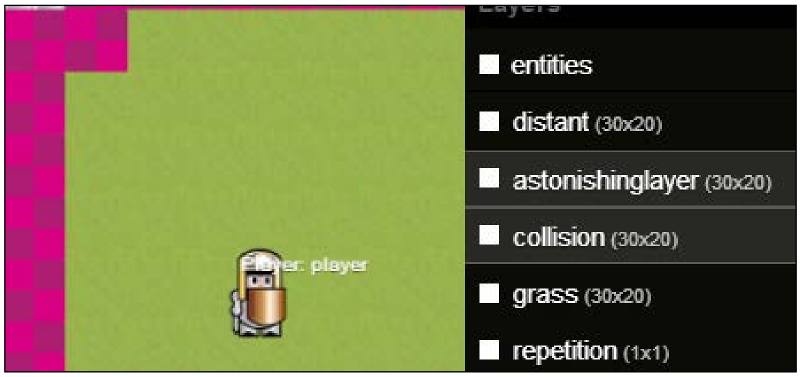
The **Repeat** option is used for background layers. For example, your background clouds can be repeated over and over again if they are a pattern.

Finally the **Link with Collision** option will make sure that, for every object you draw, collision squares are added to the **collision** layer. You can delete them from the **collision** layer later on, but it is a useful tool to speed up the drawing of walls and other impassable terrain.

Layers can be rearranged within the **Layers** menu by dragging them up or down the list. By dragging a layer to the top or down to the bottom of the list, you define its place on the z axis. You should look at the z axis as the level's third dimension, just as the world we live in has an x axis (width), a y axis (height), and a z axis (depth). The game you build is not really 3D in the conventional sense, but since the 2D graphics are layers stacked on top of one another, there is an actual third dimension at work here. The graphical layer at the top of the list will always be visible and will even hide entities. The bottom layer can only be visible when nothing else is on the way. The **collision** layer is never visible but dragging it to the top will enable you to make modifications to it more easily.

Try rearranging the layers and check out what happens. Save your game and reload. Depending on what crazy stuff you did with the layers, the world is now a very different place indeed.

Instead of dragging a layer to the top of the stack in order to be able to view it, you can also turn layers on and off. This is done by clicking on the square in front of the layer name. This will not have any effect in the actual game; it is visible only in Weltmeister. This is very useful for the **collision** layer. Try dragging the **collision** layer to the top of the stack and turn it on and off at your will. You will notice that this is the best place for the **collision** layer to be while working with the Weltmeister. This is because the **collision** layer itself doesn't actually have graphics while playing the game, so it can't obscure anything else.



Summing up the details we came across, we conclude that:

- A level is made up of different layers with properties such as tilesize, distance, and whether it is a **collision** layer
- Add a new layer using the (**+**) sign in the **Layers** menu and give it the name `astonishinglayer`

- Add a tileset `media/tree.png` to the layer. Set its dimension to `30 x 20` and its tilesize to `32`
- Try messing around with all the properties you can find on the layer including dragging the layer up or down
- Save the level and reload your game in the browser every time you adjust a parameter

Adding and removing entities and objects

There are three big types of layers: **entities**, **collision**, and any other layer. For entities and dead objects, the entities and graphical layers are of interest.

The **entities** layer holds all the entities that are present in the `entity` folder and are called upon by the `main.js` script. An entity can be anything, from the character the player uses, to an invisible trap that kills everything that dares to come close. All functionalities and the level's AI are in these entities. It can hold enemies, triggers, level changes, randomly flying objects, fireable projectiles, and everything that can be interacted with.

If you have critical bugs in these entities or some non-existing ones are included in your `main.js` script, the Weltmeister will not even load. So make sure at all times that these entities are bug-free (or not included) when you want to build a level.

Some entities, such as the player, are already present in the level. First select the **entities** layer in the **Layers** menu and then select the player entity in order to see its properties. The **x:** and **y:** properties are its current location and are always present while putting a new entity in the level.

Try moving the player entity around by selecting him and dragging him somewhere else. The **x:** and **y:** coordinates now change.

Let's add an **enemy** entity to the level. Select the **entities** layer and press the Space bar while your mouse hovers over the level. A menu will appear next to your mouse; select the **enemy** entity in this menu. An enemy just appeared at your mouse's location! You will now be able to go crazy and basically paint every square with enemy entities, but this might be a little bit of overkill, so let's just place one enemy for now. Save and reload your game. Now tremble in fear as your enemy attacks you or stare at it unimpressed, your choice.

If you added too many enemies to safely roam about, remove them from the game by first selecting the **entities** layer in Weltmeister, then the enemies you want to get rid of and simply press the *Delete* key.

It's a good habit to have both the game and the Weltmeister open to check the changes you made. If, for some reason, an entity you added is corrupt and the game refuses to load, at least you know the problem lies in the last changes that you made. Of course, you still have the Chrome or Firefox debuggers that will also point you in the right direction.

Adding objects is different from adding entities. Dead objects, which cannot be interacted with but are just a graphical thing, can simply be painted, for example, a square patch of grass, a fountain, or a castle wall. Complex interactions with these objects can be done but only with the use of entities. Here we will have a look at how to add a simple object, without interaction, to a level.

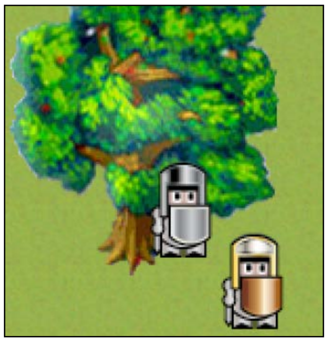
Although the level looks pretty neat, we will need to give it a makeover. Let's select the grass layer from the **Layers** menu. Hover your mouse over the map and press the Space bar key. A tileset will appear; you can make it disappear by tapping the Space bar key again. If this tileset does not fit your screen you can either hover your mouse to a more central location and bring it up there or zoom out with the scroll wheel of your mouse. If you do not have a scroll wheel, you can use the *Ctrl* + - (minus) key combination to zoom out and the *Ctrl* key with the plus sign key (+) to zoom back in. Now you can see your entire grass tileset. Select the grass and start painting it everywhere by clicking and holding the left mouse button.

A little trick for painting large areas with a single tile is to first paint only a small area on the map. Then click on the *Shift* + left mouse button and select this freshly drawn bigger area of tiles from the level itself. You can now paint with this new selection of tiles and cover more area in less time.

If you want to delete something from a given layer, simply select an empty square that is empty for that particular layer. If you already have graphics from other layers on a certain spot, but not from the layer you are currently working on, that square can be considered empty. Now paint with this empty square and the formerly chosen tiles will magically disappear. Try deleting some of your grass now.

Grass is at the bottom of everything. If you have an object, any object, it will always be on top of the grass, never underneath (except maybe in some crazy mole world). To make it so, you must drag your grass layer to the bottom of the layer stack.

Let's add something else to the scene. We still have the layer that we created, `astonishinglayer`, ready to go, so let's draw a tree with it. In order to select the entire tree at once, select the tree by clicking on the *Shift* + left mouse button key combination. Depending on where you have put your layer, the tree will now always appear either in front of or behind the player. If you dragged the layer to the bottom of the list, it might even be invisible. It's a strange result and we will deal with this later. Save your level and reload to check out your first level creativity.



Summing up the process of adding and removing entities and objects, we conclude that:

- The entity layer provides the choice of all your game entities
- You can add some of the present entities to the level, and then save and reload the game

The collision layer

The **collision** layer is a special layer that is not predefined when you open the Weltmeister from scratch. It is special because it is an invisible layer that marks impassable areas. For example, if you draw a wall on your map by using a graphical layer, all your entities will be able to go right through it as if it isn't even there. If you want a wall that is actually capable of stopping the player and his enemies, draw a line in the **collision** layer where the wall is.

Your game is still open; try drawing a wall (or any other object) and then running through it at the bottom of the level. You should find it peculiarly easy to just stroll through something that looks this solid. Select the **collision** layer, drag it to the top of the list if it isn't already done and make sure it's **visibility** option is turned on. All tiles are now clearly visible and as you can see, there are none at the bottom wall. Hover your mouse over the level's canvas and press the Space bar in order to bring up the collision tileset. Select a square and draw a line where the wall is. Deleting collision blocks is just like deleting graphics. Select an area on the map (with or without holding the *Shift* key) where no collision blocks are present and use this selection to delete the ones that are present. Save the level and reload the game. Now try walking through the wall again; it has become quite impossible; hurray for that!

Summing up the previous process:

- Select the **collision** layer in the Welmeister
- Draw some tiles with it
- Save and reload the game to see what happens if you want to walk where you drew the collision tiles

Connecting two different levels

Now that we have a sense of how to build a level by adding some graphics such as grass, trees, a player, and some enemies, it is time to have a look at how levels get connected.

For this, load the inside level into the Weltmeister. The inside level is situated inside a building (didn't see that coming, did you?). As we had to do with `myfirstepiclevel`, we need to change the call to the `loadlevel()` function in the `main.js` script, as shown in the next code snippet. This time, however, the level itself is already included in the `main.require` script.

```
this.loadLevel(LevelInside);
```

Again, don't forget the capital letters.

Load both the Weltmeister and the game itself to see if everything has been set up correctly.

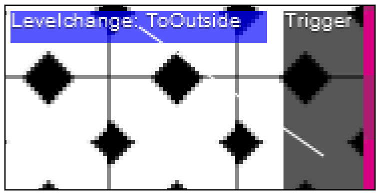
In the Weltmeister, have a look at the level's entities by selecting the **entities** layer. If you can't get a good view on the entities present in the map, feel free to turn off the other layers by clicking on their white square. Alternatively you can press the Space bar while hovering over the map to bring up the entities selection menu. As always, we have a player entity so we can move around the place, but in the menu you should notice some extra entities such as **Void**, **Trigger**, and **Levelchange**:

- The **Void** entity is a rather simple one; it is nothing but a box with a name and some coordinates
- The **Trigger** entity will trigger the code of any other entity to which it is linked if a certain type of entity (such as the player) collides with it
- The **LevelChange** entity will make the game load another level

By intelligently combining these three entities you can connect levels, so let's do just that:

1. Make sure the **entities** layer is one of the top ones, so you can see what you add.
2. Start by selecting the **Trigger** entity and place it on the map near the door. It will be nothing but a small square at first, so make it a bit bigger so as to fit the exit. You do this by selecting the box, moving your mouse to one of its edges until you see a double arrow (double arrow symbol), and dragging it to make it bigger (the same way you would resize any window object on your PC). In choosing your size, your goal is to detect the player when he wants to use the door to get out.
3. Now add a **Levelchange** entity. If you select the **Levelchange** entity, you will see its properties on the right-hand side. For now, this is just its location on the map (x and y coordinates) and its dimensions, in case you reshaped the box. Give the **Levelchange** entity a name by typing `name` in the key box and **ToOutside** as its value. Press the *Enter* key to confirm. Now you will see that the entity has an extra attribute (a name) with the value of **ToOutside**. Only by giving it a name can it be uniquely identified and that's what we need. We also need to tell him what level is needed to be loaded. Add the key **level** with a value **outside** and press the *Enter* key.
4. The **Trigger** and **Levelchange** entities are now both in the level but they are unaware of each other's existence yet; that's pretty important if we want them to cooperate.

5. Return to the trigger entity and give it a target. You do this by typing `target.1` as a key and **ToOutside** as a value. Notice the dot (`.`) after the word `target`; without it, it will not work. Now press the *Enter* key and watch as the two pretty squares get linked to each other by a white line as shown in the next figure. The **Trigger** entity now knows it is the **Levelchange** entity; it will have to trigger when touched by a player.

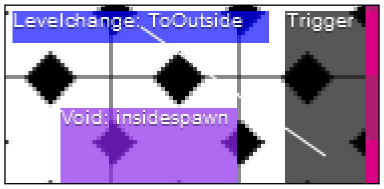


Save this and load the level. Walk your player towards the trigger location; the location of your **Levelchange** entity is irrelevant. If everything goes well, you should now be able to move to the next level by walking up to the door!

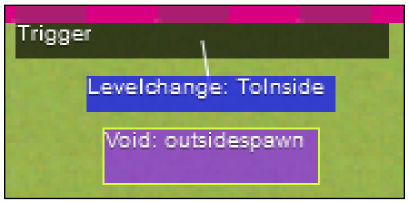
Strangely enough, when you enter the outside world, you are not placed next to the building. That is simply odd, even for a video game. Also, there is no way to go back inside when reaching for the door, you are stuck outside forever, or until you reload.

This is because no **spawnpoint**, **Trigger**, or **Levelchange** entities were added to the outside level. We will make up for that, but let's first add a spawn-point to the inside level.

To do this we will need the **Void** entity. Add the **Void** entity to the level and put it in front of the door, but past the trigger. Putting it too close to (or on top of) the trigger will cause the player to be zapped back outside. Although it's fun to make an eternal loop that zaps the player back and forth between levels, eternal loops (like dividing by zero) have a chance of destroying the world. Name the **Void** entity `insideSpawn`. Select the **Levelchange** entity and add the key **spawn** with value **OutsideSpawn**.



We are done with the inside level but now need to set up the outside level as its mirror opposite. So again, add a **Void**, **Levelchange**, and **Trigger** entity. Name the **Void** entity `OutsideDoor` since the **Levelchange** entity will look for this. Name the **Levelchange** entity `ToInside` and target the trigger to it. Also add the **Level** and **spawn** properties to the **Levelchange** entity. The values for these are (as you probably guessed) **Inside** for the **Level** property and **InsideDoor** for the **spawn** property.



Save and reload the game. If everything goes well you should now be able to move between both levels like a real pro.

Summing up the complete process of connecting two levels:

- Load the level inside in the Weltmeister
- Add three entities to the level, **Trigger**, **Levelchange**, and **Void**
- Give every entity a name
- Make the trigger point to the **Levelchange** entity
- Add this information to the **Levelchange** entity: the level it needs to load and the spawnpoint it will have to use
- Save inside, load the level outside, and repeat the exercise there
- Make sure both levels are saved and reload the game in the browser

Objects – playable and non-playable characters

Now that we have had a look at how to build a level, it is time to zoom into the code behind the entities we have been playing with. Though there is no official classification, things can be simplified by distinguishing three types of entities: dead objects, non-playable characters, and the player entity itself. These three types of entities are ordered in rising level of complexity and interactivity. In the first part of this chapter we took a look at the graphical layers of a game. Pure graphics have no element of interactivity at all; they are just present as stable elements. To get a bit of feedback from the game you are playing (building), you need entities. The least complex of these entities are the dead objects, which have no artificial intelligence whatsoever but can be interacted with, for example, pick-up items such as coins and potions. A type of entity we have already investigated is the **Trigger** entity, which itself is invisible but can be placed in the same level as graphics and can indicate what will happen further in the game. The graphics of lava will not kill you. An entity carefully placed underneath the lava, which tells the game to destroy everything that enters that area, can certainly kill you. A little bit up the scale of complexity are the **NPC's** (**non-playable characters)**. These are your enemies, your friends, everything you as a player will kill or defend, or just ignore if you feel like. They can range from mindless zombies to complex and very calculated opponents such as a chess computer. The last and most complex entity of the game is you, or at least your avatar(s). The playable character is by far the most versatile one and well worth elaborating on later in this chapter. Before we do so, we will first have to take a look at what makes an ImpactJS entity what it is.

The ImpactJS entity

In order to explain the basics of entities, it is best to first have a look at the dead objects. These entities have no complex behavioral patterns like non-playable characters or the player, but are certainly more complex than plain graphics.

An example of this is the **Void** entity, a good friend we encountered while setting up the level transition earlier in this chapter. Open the `void.js` file in your script editor so we can have a look at it. The following code snippet is an example of the **Void** entity:

```
ig.module(
  'game.entities.void'
)
.requires(
  'impact.entity'
)
```

```
.defines(function(){
  EntityVoid = ig.Entity.extend({
  _wmDrawBox: true,
  _wmBoxColor: 'rgba(128, 28, 230, 0.7)',
  _wmScalable: true,
  size: {x: 8, y: 8},
  update: function(){}
});
});
```

Every entity will at least call the `ig.module`, `.requires()`, and `.defines()` functions.

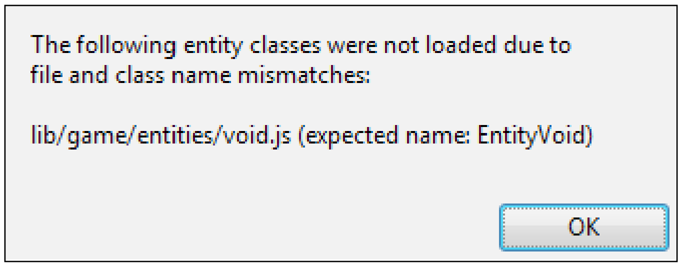
In the `ig.module` function, you define the **Void** entity as a module. The `ig.module` function call defines the **Void** entity as a new module. The module name should be the same as the script's name. The `void.js` file put in the `entities` folder within the `game` folder becomes the `game.entities.void` file.

The `requires()` function will call the code on which this entity depends. Like all entities, the void entity depends on the entity prototype code from the Impact Engine, hence the name `impact.entity`.

The `defines()` function enables you to define what this specific module is all about. Take a look at what is inside the `defines()` function. We see the `EntityVoid` module being defined as an extension of the entity class as follows:

```
EntityVoid = ig.Entity.extend({
```

Always add `Entity` in front of the entity name and don't forget the capital letters. Weltmeister will not like it if you don't and you will get an error message saying that it is expecting an entity with a different name. The following error will be generated by Weltmeister:


The following entity classes were not loaded due to
file and class name mismatches:

lib/game/entities/void.js (expected name: EntityVoid)

The **Void** entity is a special entity because it is invisible in the game; this is apparent from the fact that the code does not point to a certain image from the `media` folder. Instead it has three properties that apply to the Weltmeister: `_wmDrawBox`, `_wmBoxColor`, and `_wmScalable`. The `_wm` prefix property indicates they are important for the Weltmeister.

```
 _wmDrawBox: true,
```

The previous code snippet tells Weltmeister that it has to draw a box when the entity is inserted into a level. By setting this property to `false`, the color from the `_wmBoxColor` property will not be applied.

```
_wmBoxColor: 'rgba(128, 28, 230, 0.7)',
```

The previous code snippet defines the color of this box in an RGBA color scheme. For the **Void** entity, at present, the color is purple.

```
_wmScalable: true ,
```

The previous code snippet will allow you to make the box bigger or smaller. This is especially useful for things such as the **Trigger** entity, which you probably turned into a small but fairly long rectangle when previously connecting the two levels.

```
size: {x: 8, y: 8},
```

In the previous code snippet, the `size` property is the default size of the entity. Since this one is scalable, you can change it in the Weltmeister.

```
update: function(){}
```

Finally the `update()` function. Every entity calls this function once per frame, regardless of whether you explicitly mention the call to this function or not, as shown in the previous code snippet.

Try changing the default parameters of the **Void** entity and reload the Weltmeister to see what happens.

The **Void** entity is a simple and useful one, but let's face it, it's rather boring too. Let's have a look at something more interesting, such as coins. Let's say you want your player to grow richer every time he or she picks up a coin.

The following is a **Coin** entity example:

For this you will need a **Coin** entity, so let's open the `coin.js` file in the editor. Similar to the **Void** entity, it has a name (coin), it requires the `impact.entity` library, is an extension of the prototype entity, and has a size. There are, however, some additional interesting properties in the following code:

```
collides: ig.Entity.COLLIDES.NEVER,
type: ig.Entity.TYPE.B,
checkAgainst: ig.Entity.TYPE.A,
```

The `type`, `collides`, and `checkAgainst` properties are all about the coin's behavior regarding collision with other entities. The `type` parameter tells the game that a coin belongs to type `B` when it has to evaluate collision. The coin never actually collides with anything because its `collides` property is set to `NEVER`. Other possibilities here are: `LITE`, `PASSIVE`, `ACTIVE`, and `FIXED`. The `LITE` and `PASSIVE` entities don't collide with each other. The `FIXED` entities cannot be moved and `LITE` entities can be moved by the `ACTIVE` entities. In case an `ACTIVE` entity collides with another `ACTIVE` or `PASSIVE` entity, both entities move.

At first it sounds tricky but it is worth experimenting with. Open the `player.js` file and make sure the `collides` property is set to `ACTIVE`. Now add a coin to the game, close to the player's starting point, using the Weltmeister. Put the `checkAgainst` property of the coin in comments by adding two dashes(//) as shown in the following example:

```
//checkAgainst: ig.Entity.TYPE.A
```

If you set the mode of the **coin** entity to `FIXED` you will be unable to move the coin. When you set its mode to `PASSIVE` or `ACTIVE`, you can move the coin but it will be difficult because the coin pushes back. However, a **coin** entity set to the `LITE` property will be very easy to move about. Finally when the **coin** entity is set back to the property `NEVER`, the player goes straight through the coin as if it isn't there. We used the Weltmeister to add collision tiles to a wall; these tiles can be regarded as `FIXED` and thus will not be moved by an entity.

Remove the dashes from the `checkAgainst` property for it to work again because this tells the **coin** entity to check if an entity of type `A` touches it (the player entity is set to `A`).

While the **Void** entity is visible, the coin has in-game graphics and they are in an **AnimationSheet** frame.

```
animSheet: new ig.AnimationSheet('media/COIN.png',16,16),
```

This `AnimationSheet` frame, however, is nothing but a 16-pixel square image, so it isn't really capable of real animation. For this you would need a single PNG file with at least two different images.

We can, however, replace this coin with the second one. Do this by changing `COIN.png` to `COIN2.png` (save and reload).

The `init()` function of every entity will define their standard properties.

```
init: function(x, y , settings){
  this.parent(x,y,settings);
  this.addAnim('idle',1,[0]);
}
```

Since a **coin** entity doesn't have many properties, the `init()` method is rather empty.

What we do have is a call to the parent entity, which in this case is just `entity`. The `this.addAnim()` function is an impact function capable of animating the coin. It has three inputs:

- The status of the entity (`idle`)
- The speed between switching from one animation to another (`1` second)
- The images on the tileset it has to go through (image `0`)

Obviously since there is only one image, there is no real animation going on.

The `check()` function is a very interesting method for every entity. The following example code explains the `check()` function:

```
check: function(other){
  ig.game.addCoin();   // give the player a coin when picked up
  this.kill();     //disappear if you are picked up
}
```

It checks whether there is an overlap with another entity and, if so, will do what is stated within the function. The `check()` method is tied to the `checkagainst` property; the only relevant overlap will be the ones with the type of entity stated there. In this case the `check()` function will go off when the player touches the coin. This will result in firing the function `ig.game.addCoin()` and then removing the coin from the game with the `this.kill()` function.

Dead objects are often very simple entities with just a few lines of code, non-playable characters with even a simple AI, and playable characters are a different ballgame altogether.

Summing up the creation of playable and non-playable characters, we can conclude that:

- Contrary to pure graphics, the ImpactJS entity is an interactive game element.
- Dead objects are the least complex entities; two examples of this are the **Void** and **coin** entities.
- The **Void** entity is invisible in the game but is visible in the Weltmeister because of its special Weltmeister properties. We used it as a spawn-point previously in this chapter.
- The **coin** entity is visible in the game because it has an animation sheet. It can also be picked up by a player because of collision detection.
- Collision detection can take on several forms: entities can kill, block, push away, or simply ignore each other depending on its collision properties.
- Try messing around with all the explained parameters in the **Void** and **coin** entities to see what happens.

Setting up player controls

Nothing is more interesting than the actual player and the enemies he or she sends to oblivion.

If you open the `player.js` and `enemy.js` files, you will see there is quite a lot to discuss about these entities. From animation over controls to sound effects and beyond, they are complex indeed. All these things will be gradually unraveled during the remaining pages of this chapter. But first things first, how does ImpactJS know the difference between a playable and non-playable character?

The fact that you called an entity player does not automatically make it so; ImpactJS does not have a reserved name for this entity in order to recognize what can be controlled and what is not player-controlled. This would be very limiting indeed since **RTS** (**Real Time Strategy**) games depend on the capability of moving different playable objects at the same time. This means the only element that differentiates the two entities is whether they are controllable.

Open the `player.js` file and scroll to the following code:

```
if(ig.input.state('up') || ig.input.pressed('tbUp')){
  this.vel.y = -100;
  this.currentAnim = this.anims.up;
  this.lastPressed = 'up';
}
else if(ig.input.state('down') || ig.input.pressed('tbDown')){
  this.vel.y =  100;
```

```
          this.currentAnim = this.anims.down;
          this.lastPressed = 'down';
        }
        else if(ig.input.state('left') || ig.input.pressed('tbLeft')){
          this.vel.x = -100;
          this.currentAnim = this.anims.left;
          this.lastPressed = 'left';
        }
        else if(ig.input.state('right')||ig.input.pressed('tbRight')){
          this.vel.x = 100;
          this.currentAnim = this.anims.right;
          this.lastPressed = 'right';
        }
```

In here, we can see the player entity will react to input. When the input command `up` is given, the avatar will move upwards and show an animation. These `up`, `down`, `left`, and `right` states are not ImpactJS keywords. They are in fact defined in the main script. Open the `main.js` file and take a look at the following code:

```
        if(!ig.ua.mobile){
        ig.input.bind(ig.KEY.UP_ARROW, 'up');
        ig.input.bind(ig.KEY.DOWN_ARROW,'down');
        ig.input.bind(ig.KEY.LEFT_ARROW,'left');
        ig.input.bind(ig.KEY.RIGHT_ARROW,'right');
        // fight
        ig.input.bind(ig.KEY.SPACE,'attack');
        ig.input.bind(ig.KEY.CTRL,'block');
```

Here you can see which key is being associated to which input state. Also notice the `if` statement before the key bindings. First a check is done on whether you are dealing with a mobile device or not. This is because keys, such as the Space bar and the directional arrow, do not exist on iPads and iPhones. Try binding the attack state to your left mouse button instead of the Space bar with the code snippet as follows:

```
    ig.input.bind(ig.KEY.MOUSE1,'attack');
```

All the possible combinations can be found on the ImpactJS website.

Save and reload the game and notice how your trigger finger has moved from the Space bar to your left mouse button.

Notice how these initial key binding definitions are in the `init()` function of the `main.js` script while it is waiting for input in the `player.js` script within its update function. This is because the actual binding of keys needs to be done only once (at game initiation), while your player needs to be controlled at all times. The `update` function is called every time the game goes through a full game cycle, which is the same as your framerate. Let's assume you have a framerate of 60fps (60 frames per second); the update function will, in this case, check for user input 60 times per second.

Things are a bit different when dealing with mobile devices. Since almost no keys are present, you will need to add artificial buttons by using HTML objects.

Open the `index.html` file and type the following code for adding artificial buttons:

```
if(ig.ua.mobile){
  // controls are different on a mobile device
  ig.input.bindTouch( '#buttonLeft', 'tbLeft' );
  ig.input.bindTouch( '#buttonRight', 'tbRight' );
  ig.input.bindTouch( '#buttonUp', 'tbUp' );
  ig.input.bindTouch( '#buttonDown', 'tbDown' );
  ig.input.bindTouch( '#buttonJump', 'changeWeapon' );
  ig.input.bindTouch( '#buttonShoot', 'attack' );
}
```

When loading an ImpactJS game into the browser, it is this page that is actually loaded and the game itself is only shown in a canvas element within the page. This means that next to this canvas element, other things can be added, such as HTML buttons. Since every button can be pressed with a touchpad, an unlimited number of interactive functionalities can be added to the game by the smart use of these buttons. You can find the following button definitions in the `index.html` file, as shown in the following HTML code:

```
<div class="button" id="buttonLeft"></div>
<div class="button" id="buttonRight"></div>
<div class="button" id="buttonUp"></div>
<div class="button" id="buttonDown"></div>
<div class="button" id="buttonShoot"></div>
<div class="button" id="buttonJump"></div>
```

The buttons are `<div>` elements in which `div` is the short form for division. The <`div`> elements are used together with CSS code to lay out a webpage. In this case they provide us with four arrows to press for choosing the direction.

A `<div>` element has several attributes; among them, the `id` attribute is of particular interest to us since it uniquely identifies the `<div>` elements and enables us to link to the JavaScript code. This can be seen in the `bindTouch` method within the `main.js` script.

```
ig.input.bindTouch('#buttonLeft', 'tbLeft' );
```

Its first argument is the `<div>` element's unique ID preceded by a `#` symbol; this way JavaScript knows it needs to look for an ID. The second argument is the input state that we call `tbleft` (touchbound left).

If you have an iPad or any other mobile device, you would be able to load your game there if you had it on an online server.

Now the input keys (whether really on your keyboard or virtual ones) are bound to an ImpactJS state; these states can be used to follow player control. An example of this is of course moving in a certain direction.

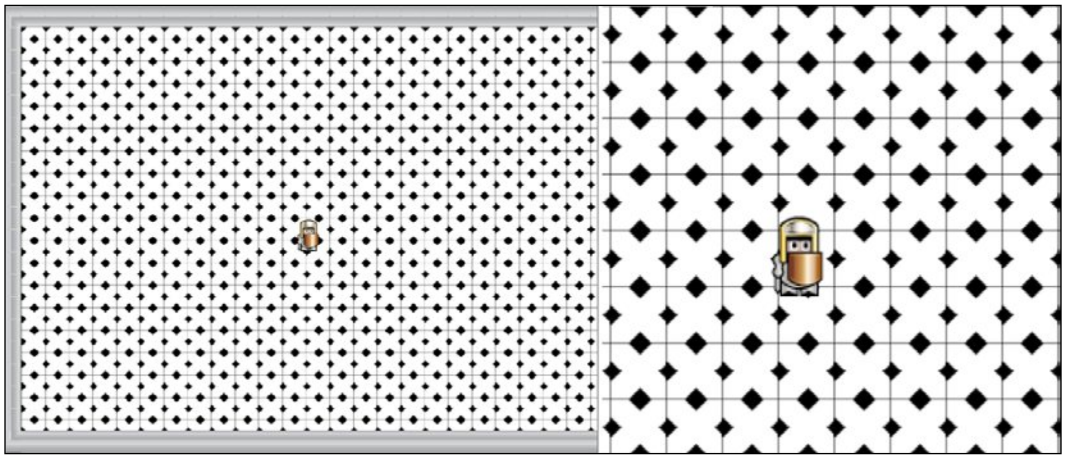
Summing up the procedure for setting up the player controls:

- Having control over an entity is what separates the playable character from the non-playable characters (NPCs).
- The link between the keyboard and an action name is defined once in the main script. You should try changing these controls to accommodate your own preferences.
- The link between an action name and actually performing the action can be found in the player entity itself.
- On mobile devices you are somewhat restricted to the touchscreen. Virtual buttons can be implemented using the HTML `<div>` tags.

Position, acceleration, and speed

Everything has a location, and some things are on route to somewhere. Positioning in the ImpactJS world is done by the x and y coordinates and a third, less intuitive, z index.

The x and y coordinates are the distances to the top-left corner of the level, measured in pixels. The x coordinate is the position of any object on the horizontal axis, counted from left to right. The y coordinate is the position on the vertical axis, counted from top to bottom. This y coordinate is a bit counter-intuitive for people who are used to looking at graphs; the y coordinate is always 0 at the bottom and higher while moving upwards. Mind you, the top-left corner of a level is not always the same as the top-left corner of the canvas! The canvas through which you can see the game is only your window on the world. This is really evident with strategy games where you never see the entire world and often get a mini map in order to navigate faster from battle to battle.



Every entity has an x and y coordinate and when you use Weltmeister, you see this change while dragging the entity over the map. Within the entity code, you can refer to (and change) its position like this:

```
this.pos.x = 100;
this.pos.y = 100;
```

This is nice if you want things to teleport around, but often you just want them to move more subtly. For this we can adjust properties such as velocity and acceleration. Setting a velocity to a number different from `0` will make the entity's position change over time. Setting the acceleration will change the velocity over time.

```
if(ig.input.state('up') || ig.input.pressed('tbUp')){
      this.vel.y = -100;
      this.currentAnim = this.anims.up;
      this.lastPressed = 'up';
}
```

We already saw this piece of code while discussing the player controls. The `this.vel.x = -100` command will make the player move upwards at a speed of 100 px per second. You need the velocity to be negative to go up because, as we saw earlier, the y axis is inversed. The speed can be set separately for every direction. For example, you can create an area where strong winds make the hero move slower when going against them but remain unaffected when moving under a 90 degree angle and the player might even move faster with the wind in his/her back with the help of this command. Try changing the speeds to simulate a strong wind from the north with the use of the following code:

```
if(ig.input.state('up') || ig.input.pressed('tbUp')){
      this.vel.y = -25;
      this.currentAnim = this.anims.up;
      this.lastPressed = 'up';
}
else if(ig.input.state('down') || ig.input.pressed('tbDown')){
      this.vel.y = 400;
      this.currentAnim = this.anims.down;
      this.lastPressed = 'down';
}
```

Acceleration in turn affects speed over time. Acceleration is a bit tricky, since slowing down does not naturally end up in stopping but in moving in the opposite direction, at which point slowing down actually becomes accelerating and vice versa. To introduce the acceleration factor, we insert the following code:

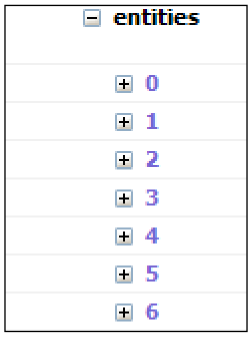
```
    if(ig.input.state('accelerate')){
      this.accel.x = 1;
      this.accel.y = 1;
    }
    if(ig.input.state('slow_down')){
      this.accel.x = -1;
      this.accel.y = -1;
    }
```

To make sure accelerating does not make your entity go at the speed of light, given enough time and button mashing, you can set a maximum velocity using the following code example:

```
maxVel:{x:200,y:200},
```

Try adding this code snippet to the `player.js init()` function or as a property. If your wind effect is still in place, it should be less strong downwind than it was before.

Apart from the x and y coordinates, a third dimension is at play. To add a sense of depth to the game, entities can be placed in front of each other. For graphical layers, this can simply be done by shifting them up and down in the Weltmeister **Layers** menu. There you could put layers permanently in front of or behind each other and all entities. However, how entities are resolved among each other is not set in the Weltmeister, but by their respective z indexes. The z index of an entity is actually its place in the array of entities. To get a better grasp of what this means, have a look at the following Firebug representation of the game's DOM:



Entities that are at the end of the array will get drawn last by the game's `draw()` method. Getting drawn last means you will be drawn on top of all the other ones and thus appear as if in front of them. All newly spawned entities get appended to the end of the list. The younger the entity, the closer it will appear when put above others. This can be avoided by manually setting the z index and using the game's `sortEntitiesDeferred()` method in the `main.js` update function of the `player.js` file:

```
zIndex:999,
```

Update the `main.js update()` function as follows:

```
ig.game.sortEntitiesDeferred() ;
```

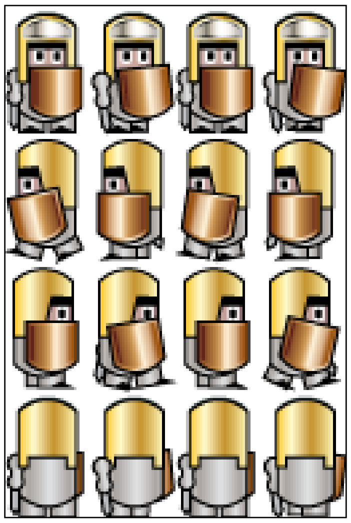
Your player can move, but how does it move so gracefully instead of just gliding from point A to B? This all has to do with sprites and animation sheets.

Summing up the position, acceleration, and speeding procedure, we conclude:

- Every entity has a position, a velocity, and acceleration.
- Try changing the player's velocity in order to change his/her position.
- Try changing the acceleration in order to change the velocity and consequently the player's position.
- Every entity has a z coordinate, which indicates whether the entity should be drawn in front of or behind the other entities. Try changing the z coordinate for the player to a very high number. Now the playable character will be drawn behind all other entities in the level.

The game's graphics: sprites and animation sheets

A sprite is a drawing, which is put on a transparent background and then saved in a file format that can keep that background transparent like a `.png` or `.gif` format. JPEG, for instance, cannot have transparent parts. Having one drawing of a character, let's say a red whale with nuclear claws, is nice. However, for animation you need more than one of these drawings, preferably from different angles. All these drawings are then put together in one file (again, not a `.JPEG` format) where they form an animation sheet.



Good sprites and animation sheets are not all that easy to come by and the ones you can find on the Internet are often licensed and prohibited for game publishing. You can either draw them yourself or buy them on websites such as `www.sprites4games.com`.

The animation sheets are typically placed in the `media` folder, though it's not mandatory and entirely up to you on how you structure them.

You assign an animation sheet to an entity by calling the `AnimationSheet()` method as follows:

```
animSheet: new ig.AnimationSheet('media/player.png',32,48),
```

The first argument is the location and name of your animation sheet. Never forget, a location is always specified relative to its root folder, which should now be the `myfirstawesomegame` folder. The fact that it is stored in the `htdocs` folder of the XAMP file structure does not count. The second and third arguments are respectively the width and height of every animation in pixels.

Now that the animation sheet is linked to the player, all the possible statuses a player can take need to be linked to a certain sequence of images. The entity's `addAnim()` method allows you to link the possible statuses to a certain sequence of images as shown in the following example code:

```
this.addAnim('idle',1,[0]);
this.addAnim('down',0.1,[0,1,2,3,2,1,0]);
this.addAnim('left',0.1,[4,5,6,7,6,5,4]);
this.addAnim('right',0.1,[8,9,10,11,10,9,8]);
this.addAnim('up',0.1,[12,13,14,15,14,13,12]);
```

At player initiation (the `init()` function), a few sequences are defined and given a name. The simplest one is `idle`. The player simply does nothing and only requires one image, which is at position 0 (`[0]`) on the animation sheet. All JavaScript arrays start at index 0 and so does the animation sheet array for ImpactJS. A 128 x 192 pixel animation sheet can hold 16 images of 32 x 48 pixels, numbered 0 to 15. The numbering starts at the top-left corner of the sheet and stops at the bottom-right corner, just like you would read the pages of this book (except maybe if you are Chinese).

Walking to the left requires only three distinct images: looking to the left, sticking out the right leg, and sticking out the left leg. While animating, looking to the left is repeated in between the switching of legs, which gives it the impression of someone walking, if the speed is set correctly. Here the speed in between switching images is set at `0.1` second, which is quiet hasty.

Try setting the speed to `100` seconds for the idle animation and `0.5` seconds for the walking animations as shown in the following example code:

```
this.addAnim('idle',100,[0]);
this.addAnim('down',0.5,[0,1,2,3,2,1,0]);
```

Notice how setting the speed for an idle animation to `100` seconds didn't affect it at all because there is no real animation, it's just one image anyway. Quintupling the time between images for walking does have a big visual impact though. The player now looks like he's floating, a bit like a ghost.

Finally, you need to update the entity property `currentAnim` with the animation needed at the time. Updating this entity property with the animation needed changes the animation sequence when speed and direction by user input are changed.

You could mess around with this too. For instance try setting the animation to right when the player walks left and vice versa. Combine this with a pretty slow animation and oh yes, you're moonwalking!

```
else if(ig.input.state('left') || ig.input.pressed('tbLeft')){
this.vel.x = -100;
this.currentAnim = this.anims.right;
this.lastPressed = 'right';
}
```

Summing up the procedure of boosting the game's graphics using sprite and animation sheets, we can conclude that:

- Every visible entity has an animation sheet. The animation sheet is a composition of all the different ways an entity can look. Try changing the animation sheet for the player entity.
- The animation sequence will tell the game which images should follow each other while a certain action is performed. Playing around with the sequence and speed of animation can create interesting effects. Try replicating a ghost or a moonwalking character using only the `addAnim()` method.

Spawning, health, and death

Every being has a beginning, life, and death. It would be kind of harsh to say you spawned from your mother's womb all these years ago. But in game terms, that is what you did.

In theory, there is no limit to the number of entities that can be spawned in a single game; in practice this is limited by performance issues, especially on mobile devices.

Let's have a look at an entity that is regularly spawned and destroyed: the projectile.

The projectile is spawned by the player when he feels his trigger finger itching. In the `player.js` update function you will find the following code:

```
ig.game.spawnEntity('EntityProjectile',this.pos.x,this.pos.y,
{direction:this.lastPressed})
```

Spawning is done by the `ig.game.spawnEntity` method. All that is really required for this method to work is the entity type and the location where it needs to be spawned. The fourth argument, which is a set of extra settings you might want to add, is optional, but is used now to tell the bullet in which direction it is fired.

Anything can spawn an entity. The same way the player spawns a projectile, the **Levelchange** entity will spawn the player. In the `levelchange.js` file you will find the following code:

```
if(spawnpoint) {
ig.game.spawnEntity(EntityPlayer, spawnpoint.pos.x,
spawnpoint.pos.y);
ig.game.player = ig.game.getEntitiesByType( EntityPlayer )[0]
}
```

What this piece of code does is it detects whether a spawnpoint is present in the level to which the player wants to travel and if so, kills the player that might be preset. In Weltmeister you can add a player entity to the level; this way you can test it separately without going through the fuss of walking through all the other ones that might come before it. This preset player entity is killed and replaced by a new one at the location of the appropriate spawnpoint. Then the `ig.game.player` variable is set to the first preset (`[0]`) player entity he finds. This last part is not mandatory but it's sometimes handy to have a direct link to the player entity.

The projectile itself doesn't really have a specified health in this case, but it can be killed with the following code:

```
if(this.lifetime <=100){this.lifetime +=1;}else{this.kill();}
```

Here, the projectile can only live for 100 frames. You could also control the lifetime of an entity with real timers or destroy it when it hits something on which it can inflict damage. Change the value from `100` to `1000` to see the projectile's range increase drastically. Alternatively you could add a new property to the projectile called `range`, and replace the lifetime check with this one. Add the range property in the `init()` function as follows:

```
this.range = 100;
```

In the check function replace the value `100` by `this.range`:

```
if(this.lifetime <=this.range){this.lifetime
+=1;}else{this.kill();}
```

Congratulations! Your code has again become a bit more readable and flexible.

The projectile can also be destroyed when it hits an enemy using the following code snippet:

```
check: function(other){
    if(other.name == 'enemy'){other.receiveDamage(100,this);}
    this.kill();
    this.parent();
  }
```

Killing an entity is simply done by calling the `kill()` method but the entities `receiveDamage()` method will also call the `kill()` method if health reaches the value 0.

So what happens in this projectile check function? If the projectile collides with an enemy, it will receive damage equal to `100` by `this` (the projectile). If this happens, the projectile is destroyed in the process.

Spawning and death are simple things in ImpactJS, health even more so. While you spawn or kill an entity with a method, health is nothing but a property you can set and change at will. In the `player.js` file you will see that the player has a health of `400` if the following code has been added:

```
health: 400,
```

Deducting health is built into the Impact Engine by means of the `receiveDamage()` method; you can increase health with the same method. Try setting the damage in the `receiveDamage()` method to a negative amount and you just invented the healing projectile!

```
if(other.name == 'enemy'){other.receiveDamage(-100,this);}
```

Summing up the complete process of spawning, heath, and death, we can conclude that:

- Every ImpactJS entity can spawn, lose, gain health, and get killed.
- Try changing the spawn position of the projectile entity to make him spawn closer or further away from the player.
- The projectile causes damage to other entities; try inversing the effect to create a healing arrow.

The camera view

When the world you explore is small and cozy, it's easy to keep an overview at all times. This does not remain the case with bigger levels and smaller screens. If it is your goal to ever release a game for a mobile phone, you must master the camera.

Your camera is nothing but your window to the world. When your world is big, you will need to adjust your window regularly to keep track of things. There are several types of cameras but the two most important ones are the free to move camera and the automatic camera.

However, before diving into the camera itself, it is best to have a look at the canvas element and the way it is set up in an Impact game.

The game canvas

If you open both `main.js` and `html.index`, you should find all the canvas code you need since this is such a high-level game component. Within the body tags of the HTML document you will find the canvas that holds the cinema screen of your game. The canvas element has an ID called `"canvas"`, which makes it possible to link it to JavaScript by using the following code:

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

In the `main.js` file you can find the `main` method of the `ig` object. This method links the entire game to the canvas by looking up its ID. If JavaScript needs to look up an HTML ID, it is always preceded by the `#` symbol as shown in the following example:

```
ig.main('#canvas', OpenScreen, 60, 640, 480, 1);
```

The `ig.main()` method has 6 parameters. The first one is the canvas ID, then the name of the game as specified earlier in the `main.js` file. The third parameter indicates the frames per second at which the game needs to run; however, this one has become obsolete and will probably be removed entirely in future versions. Nowadays, the engine itself decides the optimal framerate, so manual setting has become impossible.

The last three parameters are the width and height of the canvas and the zoom you want to use. Zoom is something peculiar because it upscales everything by the factor you put down.

A canvas with dimensions 640 x 480 and a zoom value of 1 will really be 640 x 480 pixels big and every character in it will have its original dimensions. If, however, you put the value of zoom as `2`, dimensions will be multiplied by 2 and so will everything in the game. If, for instance, you have only 640 x 480 pixels available, but you can barely see your main character, divide the dimensions by a value of 2 and set the zoom value to `2` as shown in the following code example:

```
ig.main('#canvas', OpenScreen, 60, 320, 240, 2);
```

Try setting the zoom value to `6` for extreme eye pain and blurriness.

Summing up canvas characteristics, we can conclude that:

- The game canvas is your window to the game world.
- Several elements of this window can be changed; size and zoom are the most important ones. Try changing them both in order to adjust perfectly to your own screen resolution.

Free to move camera

A free to move camera, as the name states, is free to move by the player himself. These viewports are used typically in RTS games because many things are under the player's command. For example, in the famous game Red Alert, you have dozens of tanks, airplanes, soldiers, and crazy submarines roaming about. Good players have theirs dispersed all over the map, attacking various targets at once. Camera controls in such a game are more sophisticated than the simple introduction we are going to explore here, but you have to start somewhere. Find the free to move camera's code in the `main.js` file:

```
var gameviewport= ig.game.screen;
if(ig.input.state('camera_right')) {gameviewport.x =
gameviewport.x + 2;}
else if(ig.input.state('camera_left')) {gameviewport.x =
gameviewport.x - 2;}
else if(ig.input.state('camera_up')) {gameviewport.y =
gameviewport.y - 2;}
else if(ig.input.state('camera_down')) {gameviewport.y =
gameviewport.y + 2;}
```

The screen object represents the part of your game that you can see, that is, the earlier mentioned view-port. Here, the screen is assigned to a local variable called `gameviewport` so that it can be manipulated with the buttons. For example, every time the player hits the `camera_right` button, the window is changed to the right by 2 pixels.

Summing up the camera movement procedure, we can conclude that:

- The free to move camera will adjust the window only when manually told to do so
- You can try moving the camera around in the game

Automatically following camera

Making an automatically following camera might sound a lot more difficult but it does not need to be. We can see the simple process of adding an automatically following camera in the following code:

```
var gameviewport= ig.game.screen;
var gamecanvas= ig.system;
var player = this.getEntitiesByType( EntityPlayer )[0];
gameviewport.x = player.pos.x - gamecanvas.width /2;
gameviewport.y = player.pos.y - gamecanvas.height /2;
```

Here an extra element is introduced: the canvas itself. The `ig.system` object makes sure the game loops and is also responsible for the input. The `ig.system` object is usually called through the `ig.main()` function, which we saw when looking at the canvas and thus takes the same arguments. Here it is assigned to a local variable `gamecanvas` and we need it to get the actual dimensions of the view-port we are handling. The player entity is also assigned to a local variable `player`. As you might have noticed, the first player entity is taken (index 0 of the array). So, in case there are multiple player entities, only the first will be focused upon. This makes it an automatically following camera, rather unsuitable for games with several playable entities.

The game window is constantly updated with the position of the player (for both x and y axes) and the map width is divided by 2. This last deduction is to keep the player firmly centered. Try leaving out this last part and watch what happens:

```
gameviewport.x = player.pos.x;
gameviewport.y = player.pos.y;
```

The view-port will be updated to keep the player on the screen but the player is placed on the top-left corner. It will always be on the top-left corner since coordinates for the x axis are counted left to right and coordinates for the y axis increase from top to bottom.

Summing up the procedure of creating an automatically following camera, we can conclude that:

- The automatically following camera tries to keep the player in the middle of the screen.
- You can try changing the code so that the player is kept in the top-left corner of the screen.

Adding music and sound effects

There are good games and there are truly, epically memorable games. Any game can hold itself just on a great gameplay and some decent graphics alone; you don't always need music. Minecraft is a great example of this type of game; you don't actually play it for its refreshing music. But for those who played Zelda Ocarina of Time and any Final Fantasy, you know that music is what puts the icing on the cake. It has to be said in advance, music can sometimes be a buggy thing on mobile devices. Concurrently playing two sounds is often impossible. This is something rather basic since background music and sound effects always overlap. Because of its unruly nature on mobiles and for the sake of reproducibility, we will only look at desktop versions here.

There are two main types of sound: the real music and the sound effects. The real music consists of composed songs; for modern (and expensive) games these are often orchestrated. The sound effects are the grunts of your enemies, the clashing of swords, the sound of your footsteps, and a gust of wind. If you want to get yourself some actual music, you can compose it yourself or buy it. When you need sound effects, you only need to get yourself an audio recorder and a list of sounds you need and organize a recording session with some of your best friends.

Playing background music

In the `main.js` file you should find the following code:

```
var play_music = true;
var music = ig.music;
music.add("media/music/backgroundMusic.ogg");
music.volume = 0.0;
music.play();
```

The first important element you see here is `ig.music`, which is (as you probably guessed) the object that takes care of all music. The music array forms the list of all the music you will want to use, and adding a song is done the same way you add something at the end of any array, that is, using the `.add()` method. The method needs only one parameter: the music file you want to use with its location relative to the game's root folder. You can set the volume with the volume property. The volume can range from value `0` to `1`. Of course you can set the volume to `1` as much as you want, there will be no sound if you don't activate the music. This is done with the `.play()` method. Try setting the music volume to 1 and reload the game.

Whether the player wants to hear your music or not should really be up to her or him. Let's say they are playing your game during class; you don't want them to get caught do you; that would be evil. For this purpose, you will find the following code in the `main.js` file:

```
if (ig.input.pressed('music_down')){ig.music.volume -= 0.1;}
if (ig.input.pressed('music_louder')){ig.music.volume += 0.1;}
if (ig.input.pressed('music_off')){ig.music.stop();}
```

It basically checks whether the sound buttons (which you have defined earlier) are pressed and if so, volume is increased, reduced, or just turned off entirely.

Summing up the entire process of adding music and sound effect, we can conclude that:

- Music can be added to the game in either the `.mp3` or `.ogg` format
- The `music` class is especially useful for entire tracks of music since it has several functions equivalent to a standard radio
- You can try changing the volume and turning the music on and off

Introducing sound effects

Music is a continuous thing and not really dependent on the game events (except maybe some more nervous music when your player is almost dead). Sound effects on the other hand can be added to almost anything.

Open the `player.js` file and find the following code in its `init()` function:

```
this.walksound = new ig.Sound('media/music/snowwalk.ogg');
this.walksound_status = false;
this.walksound.volume = 1;
```

Another new object, `ig.sound`, will be able to handle any sound you offer it, including background music. It is, however, still better to attribute your music to the `ig.music` object because of the extra options you have for handling music tracks. For example, with the `ig.music` object, you can shuffle your tracks (`.random`) or add a fade out effect (`.fadeOut`) if not already included in your MP3 file.

The walk sound is added as a new sound to the player entity (`this`) and its volume is set to `1`. We have a sound to add for the footsteps but it wouldn't make much sense to hear footsteps when he is not actually walking:

```
if(this.vel.x == 0 && this.vel.y == 0){
  this.walksound.stop();
  this.walksound_status = false;
}
else if(this.walksound_status == false){
  this.walksound.play();
  this.walksound_status = true;
}
```

When the player is not strolling around, all is quiet. If he starts walking again, the sound of footsteps resumes. There are many more examples for adding sound effects, but for now we will end it at this one.

Summing up the complete procedure to add sound effects, we can conclude that:

- A sound effect is a short sound that is usually only played when a certain action happens
- A sound effect will only play once by default
- You can try activating the snow-walk sound effect

Game physics with Box2D

To end the exploratory chapter, we will have a look at the physics engine of ImpactJS: Box2D. Physics engines are game engines capable of simulating many of the visible forces at play on earth like gravity and pressure forces (impact). One of the most famous games with a physics engine is, of course, Angry Birds. Physics was used in lots of games before this 2D world hit (such as Half-life and games even way before this one). Angry Birds should, however, be an example of how a simple game (combined with a considerable marketing machine) can reap enormous success.

The engine is not an invention of Dominic (maker of ImpactJS) but a port from Flash ActionScript to JavaScript. As such, a full description of all the Box2D capabilities is not available on the Impact website (as it is for the Impact Engine), but it is available on the following website: `http://www.box2dflash.org/docs/2.0.2/manual.php`.

The documentation on combining ImpactJS and Box2D, however, is fragmentary at best. You need a totally different way of thinking when building a game with physics versus one without and that is why the source code is also separate from the standard package. As mentioned in *Chapter 1*, *Firing Up Your First Impact Game*, you can get your Box2D source code from a downloadable file called `physics` when you buy ImpactJS. The folder called `Box2D` should be placed under the `plugins` folder in order to proceed.

Before diving into the Box2D code, load up a game and press the *Shift* + *F9* key combination. You are now magically teleported to the bizarre world of Box2D, where things can fly and gravity soaks everything back down. Try pushing the coins around and see how they react to well placed headbutts from different directions.

Gravity and force

If you open the `main.js` file, you will stumble upon a new game definition. This time it is not an extension of the standard `ig.game` function, but `ig.Box2DGame`. Yes, it is possible to define different games in a single file and often this technique is used for making game over screens, splash screens, and the sort using the following code:

```
BouncyGame = ig.Box2DGame.extend({
    gravity:3,
```

Right from the beginning we can define the gravity of the world as a property of the `BouncyGame` variable. Feel free to change it and watch the difference in gravity take effect in the game. Gravity does not need to be a positive force either. Try setting it to a negative number such as `-100` and you will see everything being drawn towards the ceiling.

The stronger the gravity, the more force you will need to overcome it. With a gravity value of `300` (or `-300`) your movement becomes restricted to left and right.

This can be changed in the player entity itself. Open the `boxPlayer.js` file to find a special instance of the player entity. Special, because it is not an extension of the normal player entity, but the other entity called the `Box2DEntity`, as shown in the following code example:

```
.requires(
  'plugins.box2d.entity'
)
.defines(function(){
  EntityBoxPlayer = ig.Box2DEntity.extend({
```

Also notice that we needed to include the Box2D entity.

While the normal Impact Engine makes use of velocity, Box2D uses vectors. As you might remember from physics and mathematics, a vector is a line with both direction and magnitude; let's take a look at how it is implemented:

```
if(ig.input.state('up')){
  this.body.ApplyForce( new b2.Vec2(0,-200),
this.body.GetPosition() );
}
```

For example, for going upwards you apply force with your own body on the position of your body. The force you output has a magnitude of `200` as shown in this example. We changed the value of gravity to `300` so we don't have enough to overcome it with a force of 200. Try setting its value to `500` and you will be able to gradually overcome gravity again. Set its value to `1000` and even though you will still fall like a brick, overcoming gravity by pressing the up button becomes a breeze.

Summing up the concept of gravity and force, we can conclude that:

- Box2D is a physics engine, not officially part of ImpactJS, but fairly integrated with it.
- Box2D is vector-based. All movements translate in a combination of force and direction. Gravity is just a specific case, always having a vertical direction.
- Try changing the game's gravity to make things float upwards.
- Change the force that is applied to the player when pressing the up button.

Collision impact and bounciness

While hitting another object like a coin, it can be moved by the force of the impact. You probably tried doing that already. The force the player exerts is applied to the coin and it goes flying. Eventually the coin is brought to rest again by gravity but you are free to hit it again of course.

The coin also has a certain amount of bounciness, which in Box2D is called restitution. The value of restitution can be set on a scale from `0` to `1`. Since force decreases over time, an object will never bounce back at the same speed with which it hit the wall. You can set the bounciness of the coin yourself in the `boxcoin.js` file as follows:

```
This.restitution = 1;
```

Try setting the value of restitution to `0` and see if the coins still bounce off the walls.

This was a very short introduction to Box2D. In the next chapter we will build up a small RPG from the ground up.

Summing up the collision impact and bounciness concept, we can conclude that:

- Collision between two bodies in a Box2D environment will translate in each body exerting a certain force on the other body
- A body can have a certain amount of elasticity when hitting a solid object; this is called restitution or bounciness
- You can try changing the restitution of the coin entity and observe the small difference in bounciness

Summary

The aim of this chapter was to grant a quick insight into each of the important components of an Impact game by exploring a pre-made example. We first used the Weltmeister tool to open an existing level and gain a deeper insight into how it is built up out of layers and entities. We took a look at a playable character and how it differs from a non-playable character. By adapting some entity parameters we could change things like health, movement speed, and even the way our entities look. Since in most games you can't see your entire playing field on a single screen, we took a look at a manual and automatically following camera. We added a background tune and sound effects as part of the atmosphere of a game.

Finally we took a quick peek at the Box2D physics engine. While in this chapter we have only been tweaking parameters, in the next chapter we will build a game from the ground up.

3
Let's Build a Role Playing Game

In the previous chapter we took a look at several key concepts and zoomed in on them one by one, largely disregarding their underlying dependencies. Now we will build up a game step by step. In this chapter we will take a look at the RPG, while in *Chapter 4, Let's build a Side Scroller Game*, we will dive into the side scroller game.

In this chapter we will cover:

- The RPG game format and its possible sub formats
- Building an actual level for your player to explore and connecting it to other levels
- Adding a playable entity, slayable but dangerous foes, and neutral talkative characters to the game
- Turning your player into a force to be reckoned with by adding weapons and helpful items
- Adding some depth to the player's enemies by bestowing them with basic artificial intelligence
- Keeping track of some changes in the game such as the collections of coins
- Ending the game by pitting the player against a more formidable foe

The RPG game setting

Before diving into the RPG game setting it would be good to have a look at some successful RPGs and see what we can learn from them. There are some great examples out there: **Zelda**, **Final Fantasy**, **Pokémon**, **World Of Warcraft**, **Tibia**, **Baldur's Gate**, **Neverwinter Nights**, and so on. The list is virtually endless. What made all these games so successful? Well, there is always the marketing component, but no game can reap eternal fame on marketing alone. So what was their *unique gaming proposition* for the audience they targeted? Game reviewers often divide their marks over several categories such as gameplay, graphics, sound, and so on. These are valid points, but why not have a look at addictiveness for example? Even the simplest game can be addictive. If you have ever been to Vegas and witnessed the tons of people playing for hours on slot machines that require no skill at all, you will understand that there is something special about game psychology.

Addictiveness is of course great, if you offer a free game and want to make money from in-game advertising or recurring subscription fees. Another approach is making the game engaging but final. Those are the games you can actually "finish". They usually have an interesting story between the games' protagonist and antagonist. When the antagonist is defeated, the game ends and as a player you are unlikely to pick it up again. An example would be every game within the Final Fantasy series.

Most RPGs apart from **MMORPG**s (**Mass Multiplayer Online Role Playing Game**) are in this second category. They often have a fascinating story and mesmerizing music. The characters are really interesting and deep. The battle system is very intuitive and yet complicated enough for someone to be either good or bad at it. The really good ones tend to leave a lasting impression on anyone who played them, and they cost a lot of work to put together.

This is the standard way of looking at an RPG. However, this should in no way hold you back from refreshing the genre and throwing some elements from other genres or totally new ideas into the mix. For example, **Borderlands** is a crossbreed of an RPG and a shooter. It has level progression and weapon enhancement like most RPGs. It has a story and at the same time it still plays like a shooter.

A game does not need to mix up two computer genres. **Minecraft** is essentially the pleasure of playing with Lego brought to a computer.

What it boils down to is finding out for yourself what things you enjoy the most or enjoyed greatly as a kid. Find the underlying mechanic at work and try to replicate that feeling in a game. This is all easier said than done for sure. However, it is necessary to go through this process since it is your time that it will take up to build the game, and if it isn't even a game you yourself would like to play, why would others want it?

For RPGs, it often gets more complicated than simply finding an original gameplay component. RPG video games can be a mix between a great book and a movie with the merits of interactivity thrown in. If you are a great story teller, or know one, why not do it this way? A game does not need to be difficult or graphically perfect for people to play. A good example is Final Fantasy VII, which was a big hit in the 1990s. In 2012 it was rereleased with "polished graphics". There is not much difference; an untrained eye wouldn't immediately notice the polishing. But it's still a great game, regardless of the fact that it can't compete with the complexity and graphical splendor of games such as Skyrim or Fable.

This is what you should be aiming at: take the *core pleasure* you want your game to have, package it with as little complexity as you possibly can, and add happy, pastel-colored graphics. Happy graphics are great. No, seriously, if you want your game to radiate darkness and fear, that's fine too, but otherwise, certainly consider smiling clouds and crazy looking animals.

Building an RPG level

Now it is time to put our own little RPG together. We will start our journey with a clean slate. Following are the steps to build an RPG level:

1. Let's take a copy of the freshly installed `ImpactJS` folder that we kept in the `chapter 1` folder and rename it to `RPG`. Copy the `media` folder from the `chapter 3` folder to your `RPG/media` folder. This way at least we have some graphics to work with.
2. Go back to the **it works!** screen you get when you enter `localhost/RPG` in your browser.
3. Let's start by opening the Weltmeister (`localhost/Weltmeister.html`) and drawing a small level.

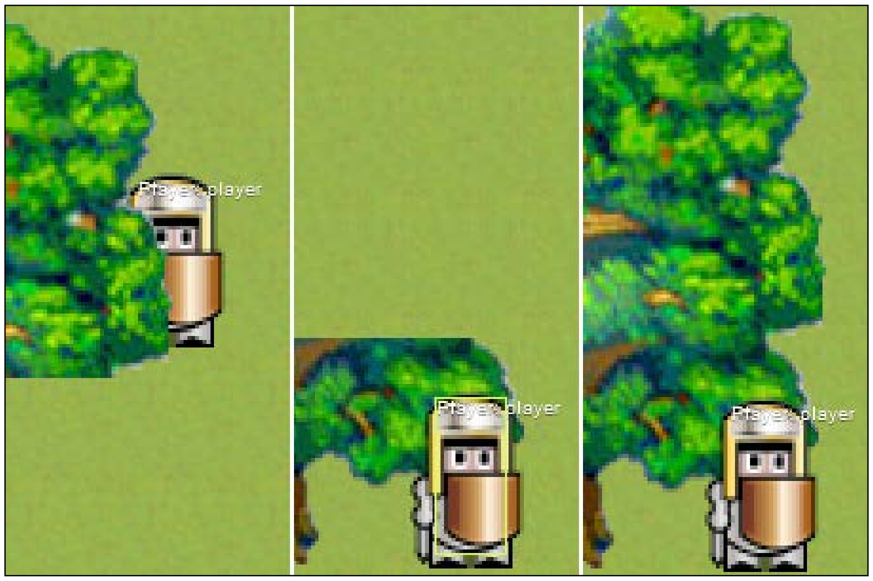
4. You will notice that there is nothing prepared for you this time around. The only available layer is the `entities` layer and it doesn't even contain a single entity. However, we can draw ourselves a small playing field to get started once we have something to populate the world.
5. So let's add another layer (+-sign) and call it `grass`. Let's set the tile size to `16` and have a 30 x 20 zone at a distance of 1 pixel. Select the tileset `grass.png` and click on the **Apply Changes** button before you can start laying down the grass.
6. If you don't have your drawbox centered just right for you, hold the *Ctrl* key and move your mouse until it is centered. If for some reason it is too big to fit on your screen, zoom out with your mouse scroll wheel.
7. Once we have painted this entire layer in green, we can easily add another one to sit on top of the grass. But before you do so, save your file as `level1`. Saving often is a virtue.
8. When adding layers you can name and use them by what they are supposed to represent. For instance, you have a layer for furniture, plants, and miscellaneous objects. This is a decent way of working, but you have to keep in mind that some layers will visually come in front of your player and monster entities, while others will appear behind them. Even something as simple as a single wall is best drawn with two layers.

Weltmeister does not support an endless number of layers. In order to keep the number of layers respectable, you can have tilesets for specific levels. For instance, you have two level settings: a city and a dungeon. Both can contain a chair, so don't be afraid to have the same chair on a tileset for your city and on a different tileset to construct your dungeon. Duplicating information will increase your overall game size but can decrease the number of layers necessary for a single level.

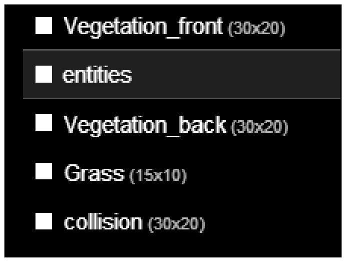
Our grass is just called `grass` because we won't have grass that will float in front of our player; consequently we don't require a second grass layer. Let's make two new layers called `vegetation_back` and `vegetation_front`. `vegetation_back` must be positioned underneath the `entities` layer in the layer selection menu. `vegetation_front` must be put above the `entities` layer. Together these two new layers will make up all the vegetation on the map.

Select the tileset `tree.png` and use the same settings for the `grass` layer.

Draw the upper part of a tree with the `vegetation_front` layer and the lower part with `vegetation_back`. The following screenshot shows the different layers:



The following are the layers you should currently have in your Weltmeister **Layers** menu:

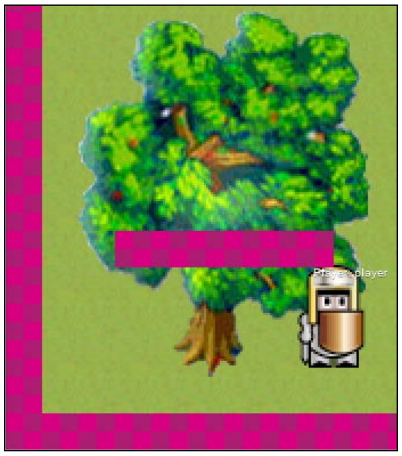


If you have no idea what the upper part or lower part of anything should be, think about how big your player and/or enemies will be. When walking past the tree, their heads or feet should not disappear. In order to avoid the player walking through the tree altogether, we will need another layer, the collision layer.

Add a layer with the name `collision` to the Weltmeister.

Don't forget, you can make a layer visible in Weltmeister by either dragging it to the top of the layer stack or switching off the layers that are blocking the view. In this case, the `grass` layer will probably block all the view if the collision layer is at the bottom of the stack. It's pretty efficient to drag the `collision` layer to the top and just turn it on and off when necessary. Settings for setting up the layer are the same as always.

With the `collision` layer, draw a border around the level so no one can escape. Also put some collision squares in the tree trunk just underneath or above the dividing line between the front and back layers, as shown in the following screenshot:



So we created a viable environment. It's not much but it's a start. However, for the level to load, we need to make changes to our `main.js` script as shown in the following code snippet:

```
.requires(
  'impact.game',
  'impact.font',
  'game.levels.level1'
)
init: function() {
  // Initialize your game here; bind keys etc.
  this.loadLevel(LevelLevel1);
},
```

In order to make sure our game finds the level, we need to include it in the `.requires` part of our module. We need to point to it in the same way as we would to any file, starting from our game root folder. The only difference is that slashes (/) are replaced by dots (`.`) and the included file itself is always considered to have the `.js` extension. For example, `/game/levels/level1.js` becomes `game.levels.level1`.

We also need to load the level at game startup, so let's add a `loadlevel()` method to the `init()` function. Don't forget that the parameter to call this function always has the following form:

`Level + Levelname` in capital letters. Anything else will crash the game.

We now have a loaded level but there is nothing interactive about it; we have no player yet. And although having **it works!** on the screen at all times is quite motivating, it's also mildly blocking our vision. So let's delete the following code from `main.js` and move on to our `player` entity using the following code:

```
var x = ig.system.width/2,
var y = ig.system.height/2;
this.font.draw( 'It Works!', x, y, ig.Font.ALIGN.CENTER );
```

Summing up the preceding content, the steps are as follows:

1. We need to start building our game from scratch in this chapter. Therefore we need the originally downloaded ImpactJS files. Put them in a separate folder in your server working directory. Also test whether you get the **it works!** message.
2. Add the `chapter 3` folder's `media` files to the folder you just set up.
3. Open the Weltmeister level editor and make a layered level. You need a collision layer, an entities layer, and three graphical layers. The bottom graphical layer will represent the grass. The other two layers represent all other objects that will show up in front of or behind the player.
4. Draw the graphical layers.
5. Include the level file in your `main` script.
6. Delete the `it works!` message from the `main` script.

Adding a playable character

In order to build our player from scratch we need a new (and empty) `.js` file. Create a new file in your code editor and even though it's empty, save it as `player.js` in the `entities` folder.

Every module starts out in the same way. It consists of the `ig.module()`, `ig.requires()`, and `ig.defines()` methods. For some modules you will not need the `requires()` method but all entities will, because here you need to include the `impact` script for entities, as shown in the following code snippet:

```
ig.module('game.entities.player')
.requires(
'impact.entity')
.defines( function(){
  EntityPlayer = ig.Entity.extend({
  });
});
```

We are going to build the player based on the `prototype` entity. This prototype has several attributes (such as `health` and `velocity`) and several methods (such as `kill()` and `receiveDamage()`) predefined. This way we only need to extend the original version with the `extend()` method in order to create our player.

There are some rules here. If your JavaScript file is called `player.js`, your entity will be called `Player`. You assign it to an extension of the `entity` prototype by adding `Entity` in front of its name, as shown in the previous code.

Any deviation from the naming conventions will remove the entity from the Weltmeister **Entities** menu. Adding an entity to the Weltmeister editor when it is correctly named, and loading the game with faulty naming will result in a crash.

Also don't forget to include the `player` entity in the `requires()` method within `main.js`. A module can only be used when the `main` module knows about its existence. The following code shows that the extension `.player` is assigned to the `entities` folder:

```
'game.entities.player'
```

If you add the `player` entity to the game with the Weltmeister right now, you would notice that there is nothing to see. The player has no visual representation yet, we address this issue in the following code:

```
EntityPlayer = ig.Entity.extend({
  size: {x:32,y:48},
  health: 200,
  animSheet: new ig.AnimationSheet('media/player.png', 32, 48 ), init:
function( x, y, settings ) {
    this.parent( x, y, settings );
    // Add the animations
    this.addAnim( 'idle', 1, [0] );
  }
});
```

In order to get a glimpse of our playable character, we need to add an animation sheet, which is located in our `media` folder. The animation sheet needs to be assigned with the right dimensions if you don't want to see just pieces of your characters walk about. We also gave the entity a size. The animation can actually be bigger than the size of an entity. If you do not set a size, you will see that you can select the `player` entity in Weltmeister, but its boundaries do not encompass the entire image. This is because the default size is 16 x 16. Size is a relevant property for collision detection. We also gave the player some health to get started. The default health is 10.

We are also confronted with the `entity` prototype's `init()` method. The `entity` prototype already has its own `init()` function, so it's best to include this by calling the `parent()` method within the `init()` function. Having an animation sheet defined does not make the entity animated. You need to assign an action to the animation sheet. Here, *idling* corresponds to the first image on the sheet. You can now safely add your player to the map.

Great, we have a player in our game! Too bad it doesn't move. Let's work on that right now.

In the `main.js` script you are to add the following to your `init()` method:

```
// move your character
ig.input.bind(ig.KEY.UP_ARROW, 'up');
ig.input.bind(ig.KEY.DOWN_ARROW,'down');
ig.input.bind(ig.KEY.LEFT_ARROW,'left');
ig.input.bind(ig.KEY.RIGHT_ARROW,'right');
```

This will make sure your arrow keys are bound to an *input state*. From now on the game will check automatically whether any of these keys are pressed. Since we are building a top-down game here, we need to be able to walk in any direction.

In the `player.js` script, four new animation sequences will need to be added to the `init()` function as shown in the following code snippet:

```
this.addAnim('down',0.1,[0,1,2,3,2,1,0]);
this.addAnim('left',0.1,[4,5,6,7,6,5,4]);
this.addAnim('right',0.1,[8,9,10,11,10,9,8]);
this.addAnim('up',0.1,[12,13,14,15,14,13,12]);
```

While the animation sequence of `idle` was composed of one image, we now need to assign a true sequence for every direction in which our player can walk. Again, the `0.1` value is the time in between images.

In addition, you will need to call and extend the `entity` prototype's `update()` function. Don't forget to have a comma to separate the `init()` and `update()` functions or you will get an error.

```
update: function(){
  this.parent();
  //player movement
  if(ig.input.state('up')){
    this.vel.y = -100;
    this.currentAnim = this.anims.up;
  }
  else if(ig.input.pressed('down')) {
    this.vel.y = 100;
    this.currentAnim = this.anims.down;
  }
  else if(ig.input.state('left')){
    this.vel.x = -100;
      this.currentAnim = this.anims.left;
  }
  else if(ig.input.state('right')){
    this.vel.x = 100;
    this.currentAnim = this.anims.right;
  }
  else{
    this.vel.y = 0;
    this.vel.x = 0;
    this.currentAnim = this.anims.idle;
  }
}
```

The `update()` function, like `init()`, is a standard method of the prototype `entity`. Therefore we need to call the parent function if we don't want to lose its ImpactJS entity core functionalities.

For every input state we need separate behavior, so we have this set of *if-then operators*. Remember that since we put this code in the `update()` function, it is run every time the game goes through an update loop, which is once per frame. The `init()` function is only called once, that is, at the moment of player creation.

Within the condition checks, we do two things: allocate a speed on the relevant axis and add an animation. If the player does nothing, the velocity in both directions is also set to `0`, so continuous input is required for the player to move.

Instead of `ig.input.state` we could use `ig.input.pressed`. But that would result in our player having to button mash his way through the level. For, every time he or she presses one of the move buttons, the player would only move a small bit and stop immediately. In case of 60 fps and a velocity of 100, the player would move 100/60 = 1.67 pixels for one touch. Although `ig.input.pressed` certainly has its merits, moving in this way might annoy even the most patient of gamers.

We finally have a playable character that moves around gracefully! It can even hide behind the tree we created earlier. We still have another problem on our hands though, we can't see our player at all times. Can you imagine the frustration of a player getting killed by something because he couldn't see where he was going? I'm sure you can and it has probably even happened to you if you have played some games in the past. However, we are in luck because a camera that follows the player around is easy to implement, as shown in the following code snippet:

```
var gameviewport= ig.game.screen;
var gamecanvas= ig.system;
var player = this.getEntitiesByType( EntityPlayer )[0];
gameviewport.x = player.pos.x - gamecanvas.width /2;
gameviewport.y = player.pos.y - gamecanvas.height /2;
```

As you can see in the preceding code, two important elements and the player are assigned to a local variable. Then the viewport coordinates are set to the player's position. If you wanted your camera to put the player in the top-left corner of the screen, you wouldn't need your game canvas. But of course, we want the player centered, so we adjust its position by half the size of the canvas in both dimensions.

On reloading the browser, you will notice that you can finally walk to the bottom of your screen and underneath the tree. Great! Just too bad that there is nothing to do here, so next we will introduce something hostile.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `player.js`.
2. Set up the `player.js` script with the standard ImpactJS module code.
3. Include `player.js` in your `main` script.
4. Add an animation sheet and sequence to your playable character so that it can be found in the Weltmeister. Also provide him with health and a size.
5. Add player controls by binding keyboard keys to input states in the `main` script.
6. Bind these input states to move the character's action by manipulating its velocity.
7. Make the movement appear as a smooth animation by introducing extra animation sequences and calling them when certain input states are active.
8. Put in a camera that automatically follows the player around wherever he ventures.

Introducing a defeatable opponent

Again, we will have to start from scratch, so open up a blank JavaScript file and save it as `enemy.js`.

The start of entity creation is always the same. Set up your `Entity` file and add an `enemy` entity to your `main` script.

In `main.js.requires` add the following code:

```
'game.entities.enemy',
```

In `enemy.js` add the following code:

```
ig.module('game.entities.enemy')
.requires('impact.entity')
.defines(function(){
  EntityEnemy = ig.Entity.extend({
  });
});
```

Adding the previous code snippets creates our entity, which we can add to the level by use of the Weltmeister. However, it's still pretty useless so let's first add some graphics using the following code:

```
size: {x:32,y:48},
animSheet: new ig.AnimationSheet('media/enemy.png',32,48),
init: function(x, y , settings){
  this.addAnim('idle',1,[0]);
  this.addAnim('down',0.1,[0,1,2,3,2,1,0]);
  this.addAnim('left',0.1,[4,5,6,7,6,5,4]);
  this.addAnim('right',0.1,[8,9,10,11,10,9,8]);
  this.addAnim('up',0.1,[12,13,14,15,14,13,12]);
  this.parent(x,y,settings);
}
```

Now we can add our first enemy to the level. It won't do much though, and you will even be able to walk right through him as if he isn't there. This is because no collision between entities has been specified yet.

Add the following code to the `player` and `enemy` entity as a property. You can add them in the `init()` function with the old JavaScript notation or above `init()` in literal notation, as shown in the following code.

The following code is for the player:

```
collides: ig.Entity.COLLIDES.ACTIVE,
type: ig.Entity.TYPE.A,
checkAgainst: ig.Entity.TYPE.B,
```

The following code is for the enemy `entity`:

```
collides: ig.Entity.COLLIDES.PASSIVE,
type: ig.Entity.TYPE.B,
checkAgainst: ig.Entity.TYPE.A,
```

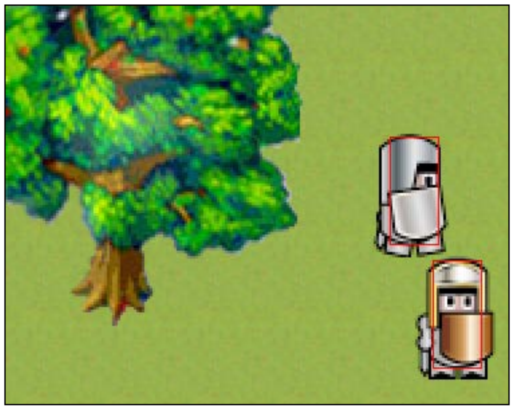
We can now push our enemy around the level like a real bully. You might have noticed that there is still some space in between the player and the enemy. This is because the boundaries of the entities are rectangles, which more than encompass the actual drawings. It's pretty annoying for a player to get hit by an enemy when visually it's not the case. To rectify the situation we need to introduce `offset` as a player property. The `size` property determines the size of the collision box around the entity. The `offset` property makes your collision box shift a few pixels to the right or down. Of course, you can enter a negative number at a point at which it will shift left and/or upward. We will need to combine these two properties to make a new collision box for the player, which makes him harder to hit. However, before proceeding it's useful to turn on the ImpactJS debugger by adding the following line of code to the `main.js` script in the `requires()` method:

```
'impact.debug.debug',
```

It's a good habit to have this debugger switched on during development. You can delete this code again when preparing it for release. Let's change the size and offset for both player and enemy using the following code:

```
size: {x:18,y:40},
offset: {x: 7, y: 4},
```

The actual image size is 32 x 48. We changed the size of both entities to `18` x `40` with an offset of `7` x `4`. If you open the debugger on the **Entities** tab and turn on **Show Collision Boxes**, you will notice the difference in size. You might also notice static collision, such as the squares of the collision layer that we added to the middle of the tree are not visible because it only shows collision for entities, as shown in the following screenshot:



There is no perfect rule for setting collision boxes. It all depends on how well centered and symmetrical your image is, how lenient you are when it comes to collision, and the difference in image size between the frontal and profile look. Here we chose to reduce our width by 14 pixels (32 - 18). In order to keep the box centered, the offset was set to half the difference ((32 - 18) / 2 = 7). The same reasoning applies to the y axis.

Now we have an enemy. Let's kill it!

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `enemy.js`.
2. Set up the `enemy.js` script with the standard ImpactJS module code.
3. Include `enemy.js` in your `main` script.
4. Add an animation sheet and several animation sequences, taking into account every direction in which your enemy might walk.

5. Change both the player's and the enemy's `collisions` entities. They need to be able to detect each other's presence so the enemy can later on damage the player.
6. If you haven't already done this, turn on the ImpactJS debugger by including it in your `main` script. The aim is to see the entity's collision boxes.

Giving the player some weapons

We do like our player to be armed and ready for some action. Let's first add a new key that will be used to attack. In `main.js` add the following key bind:

```
ig.input.bind(ig.KEY.MOUSE1,'attack');
```

In any combat situation it is the collision of two bodies that causes damage. If an arrow hits the mark, it's the arrow that damages, not the bow. The same is true for a nuclear missile. It's not the launch facility but the blast wave of the nuke that collides with whatever happens to be in the way which does the damage. In this respect you could say there are three entities at work here: a launch facility, a nuke, and its blast wave. You could even add another one if you want to make a difference between air pressure and actual conflagration. All of this is just to show how you should think when adding weapons to a game. Which impact is relevant? In case of a chicken and a chicken launcher, the chicken will become an entity while the launcher is a mere drawing.

Spawning a projectile

For our ranged attack we need a new entity, which we will call `projectile`. Make a new script, set up the basics, save it as `projectile.js`, and include it in `main.js`.

Include the following code in `main.js`:

```
'game.entities.projectile',
```

Include the following code in `projectile.js`:

```
ig.module('game.entities.projectile')
.requires('impact.entity')
.defines( function(){
  EntityProjectile = ig.Entity.extend({
    size: {x:8,y:4},
    vel: {x:100,y:0},
    animSheetX: new ig.AnimationSheet('media/projectile_x.png',8,4),
    animSheetY: new ig.AnimationSheet('media/projectile_y.png',4,8),
    init: function(x, y , settings){
      this.parent(x,y,settings);
```

```
        this.anims.xaxis = new ig.Animation(this.animSheetX,1,[0]);
        this.anims.yaxis = new ig.Animation(this.animSheetY,1,[0]);
        this.currentAnim = this.anims.xaxis;
      }
    })
  });
```

Ok, the basics don't seem to be that basic after all. This time around, we have two different animation sheets. An arrow tends to be quite a bit longer than it is wide. Therefore if the arrow is shot from left to right (or right to left), its dimensions differ from an arrow shot up or downwards. When defining an animation sheet, we have to define the dimensions that every image will take only once. However, in this case we need two different dimensions: `8` x `4` and `4` x `8`. Actually in this particular case there is another, probably easier, solution involving the angle of the animation. In programming languages there often are different ways to get the same or a similar result. However, now we will use multiple animation sheets.

We define two different animation sheets. Instead of initiating them on the standard `animSheet` property, we named them `animSheetX` and `animSheetY` to indicate the different axes. The `init()` function does not call the `addAnim()` method as in the `Player` and `Enemy` entities because it is programmed to take the `animSheet` property by default. Instead we directly address `ig.animation` to which we can pass our own animation sheets. It's nice to have an image in case you would like to add an arrow in Weltmeister, so the `currentAnim` property is given the x axis animation sequence as a default.

Now we just need to make the player spawn the arrow. Therefore we need to add the following to the player's `update()` function:

```
  if(ig.input.pressed('attack')) {
    ig.game.spawnEntity('EntityProjectile',this.pos.x,this.pos.y);
  }
```

The arrow will be spawned at the player's location.

In case you run the game at this point, the arrow can only fly in one direction: to the right. This is because our default velocity was set at `100` pixels per second to the right. Also our animation default is an arrow pointed rightwards.

That's not entirely what we want. Our enemies would at all times have to be on our right side in order for us to kill them. So let's modify the projectile code by adding the following code to the `init()` function:

```
  if (this.direction == 'right'){
    this.vel.x = this.velocity;
    this.vel.y = 0;
```

```
      this.currentAnim = this.anims.xaxis;
      this.anims.xaxis.flip.x = false;
    }
    else if (this.direction == 'left'){
      this.vel.x = -this.velocity;
      this.vel.y = 0;
      this.currentAnim = this.anims.xaxis;
      this.anims.xaxis.flip.x = true;
    }
    else if (this.direction == 'up'){
      this.vel.x = 0;
      this.vel.y = -this.velocity;
      this.currentAnim = this.anims.yaxis;
      this.anims.yaxis.flip.y = false;
      }
    else if (this.direction == 'down'){
      this.vel.x = 0;
      this.vel.y = this.velocity;
      this.currentAnim = this.anims.yaxis;
      this.anims.yaxis.flip.y = true;
    }
```

Add `velocity` as a property as shown in the following code:

```
velocity: 100,
```

What happens now is if the direction of the arrow is right, left, up, or down, it will adjust its speed and animation accordingly. There are only two images at play here, an arrow pointed upward and one pointed to the right, each in its separate animation sheet. We could add one extra image to each sheet, one downward pointing arrow, and one aiming to the left. This would be a viable solution but here we choose to use the flip property instead. Flip basically makes a mirror image of the animation, making the arrow point to the exact opposite direction. When using flip, you must make sure it actually makes sense to flip an image instead of using a separate one. For instance, if you have a character running from left to right and you want to make it run from right to left, flip is pretty ok to use. For characters running towards or away from you, this doesn't really work since you expect to either see their front or their back.

This is all very nice but where does it get its *direction* from? Let's initiate the direction with a default value and then modify the player so it can pass on its own direction to the projectile.

Add the following code to `projectile.js`:

```
direction: 'right',
```

Perform the following for `player.js`:

For every direction, add a variable called `lastpressed` with the same value as the input state, as shown in the following code snippet, for going to the right:

```
else if(ig.input.state('right')){
  this.vel.x = 100;
  this.currentAnim = this.anims.right;
  this.lastpressed = 'right';
}
```

Make the `spawnEntity` method pass the direction parameter using the following code:

```
if(ig.input.pressed('attack')) {
  ig.game.spawnEntity('EntityProjectile',this.pos.x,this.
pos.y,{direction:this.lastpressed});
}
```

Great! We now have our hero shooting arrows in every direction like a boss. At the moment, our arrows are still pretty indestructible and are quite harmless to our lucky foe. They just hit the edge of our level and stay there forever or until the game gets reloaded.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `projectile.js`.
2. Set up the `projectile.js` script. Give it two animation sheets.
3. Add the `projectile` script to the `main` script.
4. Change the player's `update` function so the player can spawn a projectile when the `attack` input state is activated.
5. Adapt the projectile's direction and animation depending on which direction the player is facing when firing.
6. Make sure the direction of the player is transferred to the `projectile` script when spawning it. This is done by filling out the optional parameter of the standard ImpactJS entity: the `spawn` function.

Causing harm with a projectile

We can make the arrow disappear when hitting an enemy or when in the air for some time using the following code:

```
lifetime: 0,
update:function(){
  if(this.lifetime<=100){this.lifetime +=1;}else{this.kill();}
  this.parent();
}
```

Initiating a new property called `lifetime` at `0` and adding a counter with the `kill()` function to the `update()` function will make the arrow disappear after flying for `100` frames. Again, don't forget to separate the `init()` and `update()` functions with a comma (`,`), or the literal notation will not forgive you.

In order to damage the enemy, we will need our arrow to check whether it encountered one. We make the arrow a `TYPE A` entity like the `player` entity, and let it check for the `TYPE B` entities like the `enemies` entity in the following code:

```
collides: ig.Entity.COLLIDES.NONE,
type: ig.Entity.TYPE.A,
checkAgainst: ig.Entity.TYPE.B,
```

By adding the `check()` function we can make the arrow check for every entity it needs to check for (as set by the `checkAgainst` property). If it encounters an entity of type `B`, that entity receives a damage of `100` as shown in the following code snippet:

```
check: function(other){
  other.receiveDamage(100,this);
  this.kill();
  this.parent();
}
```

Now we still didn't solve the problem of the arrow camping against the edge of the level or any other place where map collision is present. So let's make some bouncing arrows! No worries, we made sure they couldn't hurt the player since they will only check for entities of type `B` and will fly right through our player.

First set the `bounciness` to `1`, which means all speed is kept when bouncing back, using the following code:

```
bounciness: 1,
```

Now we only need to check if the speed has inverted (if the arrow has bounced), and invert the animation if required. Of course, this needs to be done in the `update()` function as shown in the following code snippet since it can happen at all times:

```
if (this.vel.x< 0 &&this.direction == 'right'){this.anims.xaxis.flip.x
= true;}
else if (this.vel.x> 0 &&this.direction == 'left'){this.anims.xaxis.
flip.x = false;}
else if (this.vel.y> 0 &&this.direction == 'up'){this.anims.yaxis.
flip.y = true;}
else if (this.vel.y< 0 &&this.direction == 'down'){this.anims.yaxis.
flip.y = false;}
```

This is a very naive check since it relies on the assumption that the speed of the arrow remains the same at all times, even after a bounce. However, for the sake of keeping the example simple, it will do.

We didn't even set the `health` value of our enemy and we can already damage and kill it. This is because by default the `health` value of an entity is set to `10`. Let's change this property so that our enemy can at least survive the first hit.

Make the change as per the following code in `enemy.js`:

```
health: 200,
```

Our enemy has become harder to kill but it's not as if he's a challenge for us yet. It's time to get into some basic **AI** or **Artificial Intelligence** of a non-playable character.

Summing up the preceding content, the steps are as follows:

1. Add a maximum lifespan to your projectile so it can't remain in the game forever.
2. Add entity collision detection so it can collide with an enemy.
3. Set up the projectile's `check` function so that when the projectile collides with an enemy, the projectile is killed and the enemy is damaged.
4. Add `bounciness` so it can bounce off walls.
5. Set the `health` property of your enemy so it does not get hit by the very first projectile.

Bringing your NPCs to life with artificial intelligence

Artificial intelligence can be one of the most complicated, if not *the* most complicated element of a game. As the name states, AI is artificial or simulated intelligence. The entities in your game will need to act and react to the things you as a player are doing to them or their environment. When writing AI, you are in effect trying to put the human brain or something more powerful, into the computer. For strategy games, AI can make or break the gameplay since it is what keeps the player engaged when playing skirmish matches offline. For other genres such as 2D shooters, you might be contented with enemies who do more than just shooting at you. The problem with complicated AI is that it needs to take so many parameters into account that it can become almost impossible for a single programmer to fathom. Let's make a division in three types:

- **Single-strategy AI**
- **Multiple-strategy AI**
- **Data-driven AI**

Strategies are patterns of behavior that an entity will follow in a specific situation. An enemy can charge at you with full ferocity when at full health, but retreat and look for a safe place to heal itself when badly injured. This is an example of using two different strategies while a *single strategy* enemy might just keep on attacking you until it is dead, regardless of its own life.

Data driven AI is something else altogether. It is not hardcoded behavior but requires tons of player data, which gets uploaded to a single location. There the data is processed and statistical procedures such as regressions, decision tree modeling, and neural networks are applied to make the AI more competent in the future. What you get is a learning entity, which becomes increasingly harder to defeat and automatically invents new strategies depending on the models' predictions. To some people, the thought of a computer being able to learn and adapt behavior might be rather scary. However, it is today's reality, and the future is bound to bring increasingly smarter AI. Whether computers will eventually take over the world, like in the movies Terminator and The Matrix, remains to be seen.

For now we will forget about all those data-driven statistical solutions and just have a look at a single strategy AI.

When writing AI, we want to keep a clear division between the decision and the actual behavior. You can see it as the division between the human brain and the body. The brain takes the decision and sends impulses to the body to perform the action. For this reason we will write our "brain" in a separate module, while the actions an enemy is able to perform will stay in the `enemy` entity itself as methods.

The NPC's behavior

Create a new script, name it `ai.js`, and save it under the `plugins` folder as shown in the following code snippet:

```
ig.module('plugins.ai').
defines(function(){
  ig.ai = ig.Class.extend({
  })
})
```

We kick off by defining our brand new module, our first plugin. Let's not forget to require the script in our `main.js` as shown in the following code:

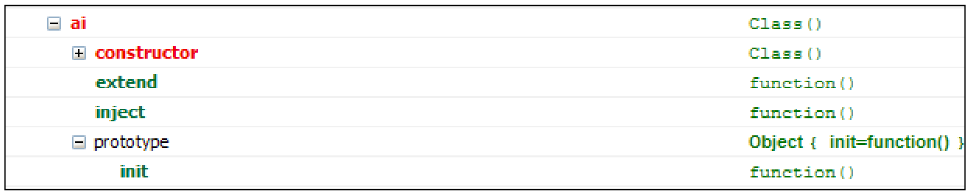
```
'plugins.ai',
```

The AI will need to give orders to the entity. For this to happen they need to speak a common tongue. Just like your legs will need to interpret your nerve signals, our enemy will need to interpret the actions it needs to perform at any given time. We define these commands in our `init()` function, as shown in the following code snippet:

```
init: function(entity){
  ig.ai.ACTION = { Rest:0,MoveLeft:1,MoveRight:2,MoveUp:3,MoveDown:4,A
ttack:5,Block:6 };
  this.entity = entity;
}
```

The `action` array holds all the possible actions the `AI` module can send. The `init()` function takes in the entity it needs to command as its input. It is not necessary to assign an entity to `this.entity` as shown in the previous code snippet (`this.entity=entity;`), but it merely confirms that `this` is not the entity itself but its AI. The fact that the input parameter `entity` is not assigned to `this` but to `this.entity` would make it possible to have a collective `ai`, capable of also making decisions for a group of enemies as a whole. This collective AI or hive mind will be addressed in *Chapter 5, Adding Some Advanced Features to your Game*.

If you would now take a look at your Firebug DOM in Firefox, you can see the `AI` class as part of the `ig` object and it currently only holds the `init()` function we just wrote. It's always a good idea to keep a track of how your DOM evolves while writing code.



⊟ ai	Class()
⊞ constructor	Class()
extend	function()
inject	function()
⊟ prototype	Object { init=function() }
init	function()

Now that we have defined the signals we will send, let's have a look at where they end up. Open the `enemy.js` script and add the following `update()` function to it:

```
update: function(){
/* let the artificial intelligence engine tell us what to do */
  var action = ai.getAction(this);
/* listen to the commands with an appropriate animation and velocity
*/
  switch(action){
    case ig.ai.ACTION.Rest:
    this.currentAnim = this.anims.idle;
    this.vel.x = 0;
    this.vel.y = 0;
    break;
    case ig.ai.ACTION.MoveLeft:
    this.currentAnim = this.anims.left;
    this.vel.x = -this.speed;
    break;
    case ig.ai.ACTION.MoveRight :
    this.currentAnim = this.anims.right;
    this.vel.x = this.speed;
    break;
    case ig.ai.ACTION.MoveUp:
    this.currentAnim = this.anims.up;
    this.vel.y = -this.speed;
    break;
    case ig.ai.ACTION.MoveDown:
    this.currentAnim = this.anims.down;
    this.vel.y = this.speed;
    break;
    case ig.ai.ACTION.Attack:
    this.currentAnim = this.anims.idle;
    this.vel.x = 0;
```

```
        this.vel.y = 0;
        ig.game.getEntitiesByType('EntityPlayer')[0].
    receiveDamage(2,this);
        break;
        default:
        this.currentAnim = this.anims.idle;
        this.vel.x = 0;
        this.vel.y = 0;
        break;
      }
      this.parent();
    }
```

We could write all the behavior in separate methods, which would then take the `AI` commands to see if they need to do something. These methods can then be put in the `update()` function of the entity in order to keep its orders up to date. We are not going to make this division in methods in this case. Because things aren't too complex in this case, all behavioral code will fit into the `update()` function without creating intermediate methods.

The `update()` function now consists of two main parts: a *call to the AI module* to receive the action it needs to perform and *actually performing the action*.

The action is stored in a local variable called `action` by calling the `ai.getAction()` method. However, in order to do this we need to add the AI to our enemy's `requires` function next to the `impact` entity code, as shown in the following code snippet:

```
    .requires('impact.entity','plugins.ai')
```

Also give your enemy a speed parameter, as shown in the following code, since the case statements use it for setting their movement:

```
    speed:50
```

All actions that we defined in the `AI` module are represented in the `update()` function. In order to make the series of case checks more efficient, a break is inserted at the end of each action. This way, once an action matches with the case it will stop checking if other cases match. We know that we want to give only *one order at every given time* so this makes sense. All code within `update()` functions should be written as efficiently as possible since it will be called 60 times per second if the game runs at a frame rate of 60. Four of our actions are geared towards moving in the correct direction, and then we have `attack` and `rest`. To make sure every situation is handled, a `default` value is set. This way, if the enemy is given a command that he doesn't understand, he will just stay put. You could rewrite the `default` part of the code and overwrite it with the `attack` case if you like; this way the enemy will always attack if he doesn't understand what he needs to do; barbaric but effective.

In case the enemy attacks, he calls the player's `receive damage` function. This is interesting since the `receive damage` method of the player can be overwritten in `player.js` to incorporate damage reduction from armor and the like.

However, for now let's take a look at the actual brain or decision making itself. Therefore we need to return to our `AI` module.

Summing up the preceding content, the conclusions are as follows:

- The AI of an entity is its ability to make decisions based on external input, often using several strategies
- In code, the decision making should be separated from the actual behavior where possible

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `ai.js`. As an analogy to the human body, this file will contain everything about the brain.
2. Set up the `ai.js` script so it becomes an ImpactJS class extension.
3. Include `ai.js` in your `main` script.
4. Define the language that will bind behavioral decisions to actual behavior. As an analogy to the human body, these would be the electrical impulses your nervous system transmits.
5. Build the actual behavioral patterns an enemy will follow for every command he can receive. As an analogy to the human body, this would be the body's reaction to certain nerve impulses.
6. Include the function that calls for an AI command. As an analogy to the human body, this function call would be the nerves themselves.

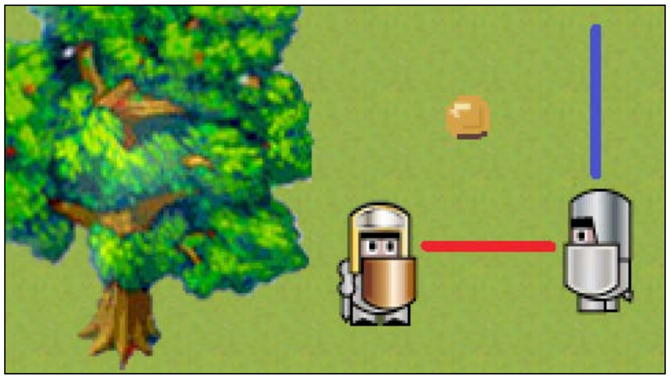
The NPC's decision making process

We just saw that the AI `getAction()` method was called but not yet fully explained. Its main purpose is to return an action when called. The possible actions here are moving in a certain direction, attacking, blocking incoming attacks, or not moving at all. What action to take is decided by the distance between the player and the `enemy` entity that needs to take the decision, as shown in the following code:

```
getAction: function(entity){
  this.entity = entity;
  //by default do nothing
  var playerList= ig.game.getEntitiesByType('EntityPlayer');
  var player = playerList[0];
  var distance = this.entity.distanceTo(player);
  var angle = this.entity.angleTo(player);
  var x_dist = distance * Math.cos(angle);
  var y_dist = distance * Math.sin(angle);
  var collision = ig.game.collisionMap ;
  //if collision between the player and the enemy occurs
  //collision.trace is the way ImpactJS simulates line of sight
detection. This will be explained after this block of code.
  var res = collision.trace( this.entity.pos.x,this.entity.pos.y,x_
dist,y_dist,
    this.entity.size.x,this.entity.size.y);
  if( res.collision.x){
    if(angle > 0){return this.doAction(ig.ai.ACTION.MoveUp);}
else{return this.doAction(ig.ai.ACTION.MoveDown);}
  }
  if(res.collision.y){
    if(Math.abs(angle) >Math.PI / 2){return this.doAction(ig.
ai.ACTION.MoveLeft)}else{return this.doAction(ig.ai.ACTION.
MoveRight);}
  }
  if(distance < 30){
    //decide between attacking, blocking or just being lazy //
    var decide = Math.random();
    if(decide < 0.3){return this.doAction(ig.ai.ACTION.Block);}
    if(decide < 0.6){return this.doAction(ig.ai.ACTION.Attack);}
    return this.doAction(ig.ai.ACTION.Rest);
  }
  if( distance > 30 && distance < 300) {
    //if you can walk in a straight line: go for it
    if(Math.abs(angle) <Math.PI / 4){ return this.doAction(ig.
ai.ACTION.MoveRight); }
    if(Math.abs(angle) > 3 * Math.PI / 4) {return this.doAction(ig.
ai.ACTION.MoveLeft);}
    if(angle < 0){return this.doAction(ig.ai.ACTION.MoveUp);}
    return this.doAction(ig.ai.ACTION.MoveDown);
  }
  return this.doAction(ig.ai.ACTION.Rest);
}
```

Add this function to the `AI` module. Just like the `init()` function, it takes the entity as an input parameter. A series of local variables is calculated to decide what path needs to be taken in order to get to the player. The enemy needs to know its distance from the player and the angle towards the player. Collision is calculated by use of the `collision.trace()` method. The inputs for this method are the entities `position`, `size`, and `distance` to its target, in this case, the player. Here you shouldn't see collision as a real physical collision but rather as a line of sight. `res.x.collision` should be interpreted as "is the player in line of sight if I look in a horizontal line across the screen?"

The following screenshot shows the line of sight for the enemy:



If this is the case, there is no more need to move up or down. The same reasoning works for the y-axis and moving left or right. This is only to show you how this function works leaving out the first two `if` statements, and calculation of the `res` variable will still give the same result because of the logic in the next two `if` statements.

A check on the distance between the enemy and the player happens after this. If the enemy is close enough to attack (this is hardcoded at `30` pixels), the enemy attacks. This cut off could be changed by reading the actual range of the enemy and using this instead of just using `30`. Also, the enemy has the opportunity to attack once per frame; this makes for 60 attacks per second. Have you ever been hit with a sword 60 times in one second? It hurts. We could lower that by increasing the chance of the enemy doing nothing at all. Changing these two things, the code could look like the following code snippet:

```
if(distance <entity.range){
  var decide = Math.random();
  if(decide < 0.3){return this.doAction(ig.ai.ACTION.Block);}
  if(decide < 0.02){return this.doAction(ig.ai.ACTION.Attack);}
  return this.doAction(ig.ai.ACTION.Rest);
}
```

Of course you would need to change the actual damage done since 2 damages might not be that impressive or challenging for a player with 200 hit points. The following code snippet shows the change in damage:

```
ig.game.getEntitiesByType('EntityPlayer')[0].receiveDamage(40,this);
```

When the distance between the enemy and the player is 300, the enemy will move towards the player. As explained earlier it uses the angle to decide what direction to go first. In all other cases, the AI advises the entity to rest. So if the player is far away, the enemy will not attack. This way you avoid being attacked by all enemies at once. If your speed is greater you can even run away.

There is one small thing left. As you might have noticed, an action is not immediately returned but sent through the `doAction()` method. The following code snippet shows how this is done:

```
doAction: function(action){
  this.lastAction = action;
  return action;
},
```

This method, which is also added to the `AI` module, is only used to store the last action that the entity has performed. You could do without this function, but it is often handy to keep track of the last move that was performed. Applications for this are not shown in this short AI tutorial.

If you were to reload the game at this point, you should have an enemy that actually tries to kill you instead of just being passive as a stone(r).

Summing up the preceding content, the steps are as follows:

1. Calling the brain to act is done with our `getAction()` function. This function takes the entity for which a decision needs to be made as an input argument, and will return a command or an action. The logic inside this function can be as simple or complex as you like. In this example, distance to the player is the most important determinant for the action that needs to be taken.
2. Use a `line of sight` ImpactJS function to determine if the enemy will be able to see the player.
3. What an AI should do is a totally subjective thing; try adding your own commands and behavioral patterns.

Pickup items to help out your player

Now that our enemy fights back, we might be in need for some extra assistance in the form of the `pickup` items and extra weapons.

One useful `pickup` item would be an instant `healthpotion` entity so that we can heal from the damage we got.

Healing your player with a potion

Let's build ourselves an entity called `healthpotion` and include it in the `main` script `main.js`, as shown in the following code:

```
'game.entities.healthpotion',
```

Include the following code in the `healthpotion.js` script:

```
ig.module('game.entities.healthpotion')
.requires('impact.entity')
.defines( function(){
  EntityHealthpotion = ig.Entity.extend({
    size: {x:32,y:32},
    collides: ig.Entity.COLLIDES.NONE,
    type: ig.Entity.TYPE.B,
    checkAgainst: ig.Entity.TYPE.A,
    animSheet: new ig.AnimationSheet('media /healthpotion.png',20,25),
    init: function(x, y , settings){
      this.parent(x,y,settings);
      this.addAnim('idle',1,[0]);
    },
    check: function(other){
      other.receiveDamage(-500,this);
      this.kill();
    }
  })
});
```

The `healthpotion` entity is a very straightforward entity. It has no real behavior except for detecting whether a player touches it and then healing the player if he or she does.

Interesting to note here is how the `receiveDamage()` method is used with negative damage in order to heal the target. This health potion is used on pickup; it does not always have to be like this and some things can be counted by use of a `gameinfo` array.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `healthpotion.js`.
2. Set up the `healthpotion.js` script with the standard ImpactJS module code.
3. Include the `healthpotion.js` script in your `main` script.
4. Add an animation sheet and a sequence.
5. Set up the `collision` entity so it can detect when the player touches it.
6. Use the `receivedamage()` function with a negative damage; this will heal the player instead of dealing with the damage. Make the `healthpotion` entity destroy itself.

Becoming rich with coins

The `coin` entity is an example of an item that we might want to keep a count of. It is almost the same as the `healthpotion` entity except for its name, animation sheet, and the `check` function, which is as follows:

```
check: function(other){
  ig.game.addCoin();
  his.kill();
}
```

Instead of healing the player, a method called `addCoin()` is applied. This function does not work yet so you can put this line of code in a comment until we change this in the section, *Keeping score for player feedback*.

First let's address another issue. If you added the `coin` and `healthpotion` entities to the game with the Weltmeister, you might have noticed that you can actually kill the `healthpotion` and `coin` entities by shooting them. If you don't like this behavior, it can be fixed by giving every entity a unique name, as shown in the following code:

```
name: "player",
```

You can check for it in the check functions as shown in the following code:

```
check: function(other){
  if (other.name == "player"){
  //ig.game.addCoin();
  this.kill();
}}
```

Now let's get our score system to work.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `coin.js`.
2. Set up the `coin.js` file with the standard ImpactJS module code.
3. Include `coin.js` in your `main` script.
4. Add an animation sheet and a sequence.
5. Set up the `collision` entity so it can detect when the player touches it.
6. When touching the `player` entity, the `coin` entity must destroy itself and call the `addcoin()` function, which sends feedback to the game info system. The function will be defined later this chapter, so turn it on when implemented.

Keeping score for player feedback

Keeping track of a number of things is all about leaving it outside the currently loaded game. This way it can be transferred across levels and even across games. Add the following to `main.js` above the `MyGame` definition:

```
GameInfo = new function(){
  this.coins = 0;
  this.score = 0;
},
```

`GameInfo.coins` and `GameInfo.score` will now keep track of how many coins we have gathered and what is our current score.

However, we do need two functions, which will actually increase these game properties. Therefore let's add these functions to the `MyGame` definition in the `main.js` script:

```
addCoin: function(){
    GameInfo.coins += 1; //add a coin to the money
},
increaseScore: function(points){
  GameInfo.score +=points;
},
```

Now you can safely get the `ig.game.addCoin()` method out of the comments without fear of game crashes. Also, we can add a call to the `increaseScore` function at the death of an enemy. To do this, we need to change the `kill` function of our enemies in the `enemy.js` script, as shown in the following code snippet:

```
kill: function(){
  ig.game.increaseScore(100);
  this.parent();
}
```

As you can see, we keep the original function by adding the `this.parent()` line of code but add our code for increasing the score just before it.

We don't need to restrict ourselves to things that can only go up. We could put a limit on the number of projectiles our hero has and keep a count of them. Add the initial number of projectiles to the `GameInfo` array as shown in the following code snippet:

```
this.projectiles = 10;
```

We are in need of two new functions, which we can add to `MyGame` like we did for `addCoin()` and `increaseScore()`. The code for adding the two functions is as follows:

```
addProjectile: function(nbr_projectiles){
  GameInfo.projectiles +=nbr_projectiles;
},
substractProjectile: function(){
  GameInfo.projectiles -=1;
}
```

Our new attack code for the `player` entity will look like the following code snippet:

```
if(ig.input.pressed('attack')) {
  if (GameInfo.projectiles> 0){ ig.game.spawnEntity('EntityProjectile'
,this.pos.x,this.pos.y,{direction:this.lastpressed});
  ig.game.substractProjectile();
  }
}
```

First we check whether we have enough projectiles and after launching one, a `projectile` entity is subtracted from our original stack.

Great! But how do we resupply? We could just create another `pickup` item for this purpose as shown in the following code:

```
ig.module('game.entities.pickupprojectile')
.requires('impact.entity')
.defines( function(){
  EntityPickupprojectile = ig.Entity.extend({
    size: {x:8,y:4},
    collides: ig.Entity.COLLIDES.NONE,
    type: ig.Entity.TYPE.B,
    name: "pickupprojectile",
    checkAgainst: ig.Entity.TYPE.A,
    animSheet: new ig.AnimationSheet('media /projectile_x.png',8,4),
    init: function(x, y , settings){
      this.parent(x,y,settings);
      this.addAnim('idle',1,[0]);
```

```
        },
        check: function(other){
          if (other.name == "player"){
            ig.game.addProjectile(10);
            this.kill();
          }}
      })
    });
```

Add some of these to your game and you will be able to shoot your way through anything like a real Rambo!

There are many other appliances for this `GameInfo` array out there, but it's up to you to make good use of it.

Summing up the preceding content, the steps are as follows:

1. Some information needs to be kept outside of the actual game so it can be used and stored after a game is finished. This overhead information is kept in a `gameinfo` array, defined in the `main` script.
2. Create the `gameinfo` array and reserve a place for storing both the number of coins that are collected and the overall score a player achieves.
3. Build the `addcoin()` and `increasescore()` functions. `addcoin()` will increase the number of coins by one when called. `increasescore()` can take a numeric input parameter, which is the score that it needs to add to the total.
4. Activate the `addcoin()` function in the `coin` entity.
5. Overwrite the enemy's `kill` method to incorporate the `increasescore()` function.
6. Using the same logic, create both the `addProjectile()` and `substractprojectile()` functions.
7. Change the `player` entity code. So it will check how many projectiles a player has before it becomes possible to fire. When a projectile is fired, a `projectile` entity is subtracted from the remaining ammo.
8. Use everything you learned about `pickup` items to make a `pickup` projectile that will replenish the player's ammo supply.

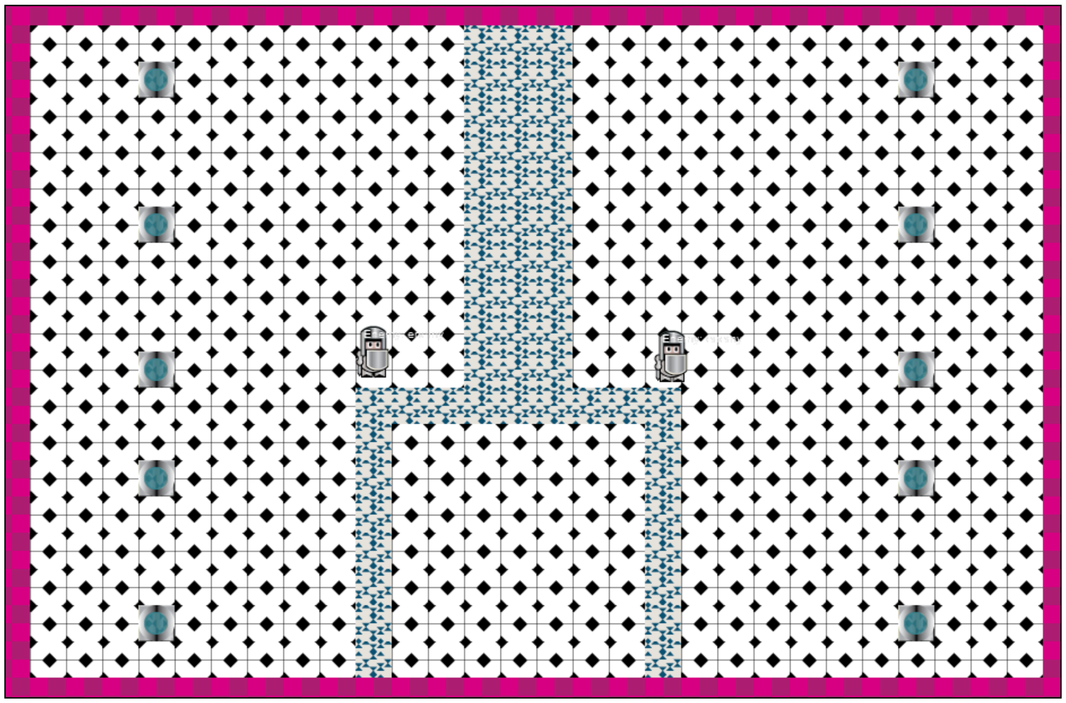
Transitioning from one area to another

Making a map transition for an RPG has been thoroughly explained in *Chapter 2, Introducing ImpactJS*. We will just have a quick recap and some pointers in this section.

As you might remember, we have used a combination of three entity files to build gateways between levels. Add the `trigger`, `levelchange`, and `void` entities to the `entities` folder and include them in the `main` script, as shown in the following code snippet:

```
'game.entities.levelchange',
'game.entities.trigger',
'game.entities.void',
```

To have a level to connect to, we need to build one first. The following screenshot shows the `endgame` level to which we should connect:



This level is the `endgame` content; it will soon feature the dangerous boss of this little RPG. Don't forget to include it in `main.js`, as shown in the following code snippet:

```
'game.levels.level1',
'game.levels.endgame',
```

Now that all necessary components are ready, connect the level in the same way as shown in *Chapter 2, Introducing ImpactJS*.

Use the `trigger` entity to trigger the `levelchange` entity when the player walks over it. The `void` entity is used as the `spawn` location.

One thing needs to be pointed out here. When the player moves from one area (level) to another, his health is reset to the default value because the `levelchange` script spawns a new player. This can be avoided by either moving the `health` value before loading the new level to an independent array of variables, or by changing the `levelchange` script itself. The second option is shown in the following code snippet. Open `levelchange.js` to find the following code:

```
ig.game.player = ig.game.getEntitiesByType( EntityPlayer )[0];
var health = ig.game.player.health;
ig.game.loadLevel( ig.global['Level'+levelName] );
if(this.spawn){
  var spawnpoint = ig.game.getEntityByName(this.spawn);
  if(spawnpoint)
  {
    ig.game.spawnEntity(EntityPlayer, spawnpoint.pos.x, spawnpoint.
pos.y);
    ig.game.player = ig.game.getEntitiesByType( EntityPlayer )[0];
    ig.game.player.health = health;
  }
}
```

Before actually loading the `level` entity, the `health` value is stored to a local variable `health`, which is then reassigned to the newly spawned player. The same can be done to any attribute, or a temporal copy can be made of the `player` entity, which then overwrites the freshly spawned one.

Summing up the preceding content, the steps are as follows:

1. Copy the `trigger`, `levelchange`, and `void` entities from the `chapter 2` folder and put them in the `entities` folder.
2. Include all three entities in the `main` script.
3. Make a level transition using these three entities as shown in *Chapter 2, Introducing ImpactJS*.
4. Change the `levelchange` entity so the player's health is temporarily stored in between level loads.

NPCs and conversation

In many 2D RPGs the epic story is told by the sole use of text. The player interacts with various **NPCs** (**Non Playable Characters**) before he or she can beat the game. An enemy is also an NPC but in most contexts, the NPC is regarded as the non-hostile character who helps the hero reach his goal by giving hints, quests, and items. We will introduce such a peaceful creature and make him speak in the next section.

The speech balloon

For this we will use a text balloon, which we treat as an entity of its own. Let's prepare a new JavaScript file and call it `textballoon.js` using the following code:

```
ig.module('game.entities.textballoon'
  )
.requires('impact.entity','impact.game'
  )
.defines( function(){
});
```

We will once again need to let our `main` script know of its existence, so add `'game.entities.textballoon'` to the `main` script.

In this file we will not only define our `textballoon` entity but also an inner class, which we will use in the `textballoon` entity: `WordWrap`. `WordWrap` is a class invented by a person who goes under the name of Kingsley on the ImpactJS forum, and all thanks should go to him. Once again this proves that looking up on forums is a good idea. Someone might have already done what you intend to do. `WordWrap` organizes the inputted text in such a way that you can fit it on objects such as speech balloons. We can define this class in any of our JavaScript files but since it's used exclusively by our `textballoon` entity, it makes sense to place the script as shown in the following code:

```
WordWrap = ig.Class.extend({
  text:"",
  maxWidth:100,
  cut: false,
  init:function (text, maxWidth, cut) {
    this.text = text;
    this.maxWidth = maxWidth;
    this.cut = cut;
  },
  wrap:function(){
    var regex = '.{1,' +this.maxWidth+ '}(\\s|$)' + (this.cut ? '|.{'
+this.maxWidth+ '}|.+$' : '|\\S+?(\\s|$)');
    return this.text.match( RegExp(regex, 'g') ).join( '\n' );
  }
}),
```

The `WordWrap` class is an extension of the general Impact class just like our `AI` module. It is in fact a function, which takes three arguments: a piece of text, a maximum width for a line of text, and whether the function should cut off by character or word. When a new `WordWrap` class is created, these three things are assigned to local arguments as can be seen in the `init()` function.

However, the most important thing is the `wrap` method of the `WordWrap` class. It only contains two lines of code but does all the work. In the first line, a regular expression is built, which is then interpreted and returned in the second line. A regular expression is a flexible way to recognize specified strings of text. How this text pattern recognition code works is not covered here since it is not within the scope of this book.

Now that we have the most vital function for our `textballoon` entity, we can build the `textballoon` entity itself using the following code:

```
  EntityTextballoon = ig.Entity.extend({
    pos:{x:0,y:0},// a default position
    size:{x:100,y:50},// the default size
    lifeTime:200,// show the balloon for 200 frames
    //media used by text balloon
    font : new ig.Font('media/font.png'),// the font sheet
    animSheet: new ig.AnimationSheet('media/gui_dialog.png',100,50),//
  the animation
    wrapper : null,// place holder
    init: function(x,y,settings){
      this.zIndex = 1000;// always show on top
      this.addAnim('idle',1,[0]);// the default graphic
      this.currentAnim = this.anims.idle;
      this.parent(x,y,settings);// defaults
      this.wrapper = new WordWrap('Epicness awaits you!',20);//we only
  have one text so use it as a default
    },
  });
```

The `balloon` entity is not much more than an image with text, which is shown on top of everything else (`zIndex = 1000`) when spawned. In the `Init()` method of our `balloon` entity, the `WordWrap()` function is used to wrap the text to the correct dimensions. It is interesting to note here how the font is initialized (`font : new ig.Font('media/font.png')`).The font that will be used is already present in our `media` folder in the `.png` format and in order to assign it to our local variable font, a new impact method is used: `ig.Font()`. Unlike the font from Word, here it has a predefined color and size. A free font tool is available if you want to make your own font for your ImpactJS game on the following link:

`http://impactjs.com/font-tool/`

There is also a variable called `lifeTime`, which will keep track of the remaining number of frames until the `balloon` entity is dismissed. This check is done in the `update()` function as shown in the following code:

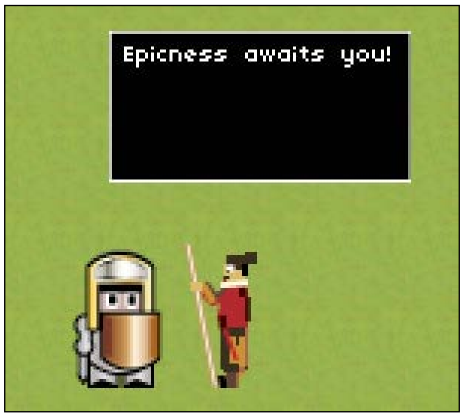
```
update:function(){
  this.lifeTime = this.lifeTime -1;// counter for the lifetime
  if(this.lifeTime< 0){this.kill();}// remove the balloon after 200
frames
  this.parent();// defaults
},
```

In every new frame, the lifetime drops by one. When the `lifeTime` value reaches `0`, the `balloon` entity is killed. More intelligent balloon timers could be implemented by counting the amount of text that should be read and adjusting the time you have to read it, but this is a simple example.

The last thing we will need is the entity's `draw()` method. `Draw()` is called for every frame just like the `update()` function, but is dedicated to what needs to be displayed as shown in the following code snippet:

```
draw:function(){
  this.parent();// defaults
  var x = this.pos.x - ig.game.screen.x + 5;// x coordinate draw
position
  var y = this.pos.y - ig.game.screen.y + 5;// y coordinate draw
position
  this.font.draw(this.wrapper.wrap(),x, y,ig.Font.ALIGN.LEFT);// put
it on the screen
}
```

All entities have a `draw` method and it is called automatically. We will have a look at it now because our balloon needs to be adjusted a bit. In the `draw()` function, first its parent function is called followed by positioning and drawing the text that needs to be shown on top of our balloon. The order of things is very important here. If you would first draw the text and put `this.parent();` at the end, the text will be written first, followed by the balloon. You can try this once we have an entity to spawn our `balloon` entity; for now you should get an empty speech balloon. The following screenshot shows a fully functioning speech balloon:



Now that we have a fully functioning speech balloon, it is time to introduce an entity that wants to talk to us: the `NPC` entity.

Summing up the preceding content, the conclusions are as follows:

- Lots of games have friendly creatures walking about and providing the player with hints.
- A talking character consists of a friendly `NPC` entity and its speech balloon, which can be regarded as a separate entity. Additionally, we make use of a `wordwrap()` function, which will keep the sentences within the borders of the speech balloon.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `textballoon.js`.
2. Create the `wordwrap()` function as an extension to the `ImpactJS` class.
3. Set up the `textballoon.js` file with the standard ImpactJS module code.
4. Include `textballoon.js` in your `main` script.
5. Add an animation sheet, an animation sequence, a size, and a default position.

6. Set the z-index property to a high number so the speech balloon is always shown on top of every other entity.
7. Use the `wordwrap()` function to transform a text of your choice and add it as a property of the speech balloon.
8. If you want to make your own font for your game, use the ImpactJS font tool to convert it into a file that Impact can use. The font tool is located at the following URL: `http://impactjs.com/font-tool/`.
9. Change the `update` function of the speech balloon so that it will keep track of how much time has passed since the speech balloon was spawned. The `update` function will also kill off the balloon when a preset number of frames have past.
10. Overwrite the default `draw` function so it will draw your text on top of the speech balloon itself.

Adding a talking non-playable character

Create a new script and save it as `Talkie.js`. `Talkie` will be the name of our lovely NPC as shown in the following code:

```
ig.module('game.entities.Talkie')
.requires('impact.entity')
.defines(function(){
EntityTalkie = ig.Entity.extend({
  })
});
```

Like for any regular entity, the `Talkie` script properties are defined before or in the `init()` function, depending on whether you wish to write them in the literal notation or not, as shown in the following code:

```
size: {x:80,y:40},
offset:{x:-5,y:0},
// how to behave when active collision occurs
collides: ig.Entity.COLLIDES.PASSIVE,
type: ig.Entity.TYPE.B,
checkAgainst: ig.Entity.TYPE.A,
name: 'Talkie',
talked:0,
Anim:'idle', times:200,
// where to find the animation sheet
animSheet: new ig.AnimationSheet('media/Talkie.png',32,48),
init: function(x, y , settings){
  this.addAnim('idle',3,[0,1]);
```

```
     this.addAnim('Talk',0.2,[0,1,2,1]);
     this.currentAnim = this.anims.idle;
     this.parent(x,y,settings);
     },
```

`Talkie` has two states, either he is doing nothing (`idle`) or he is talking (`Talk`) and his animation changes accordingly. He should only stay in the `Talk` state while the balloon is there, so a timer is set to synchronize the balloon with Talkie's animation using the following code:

```
   update: function(){
     if(this.times>=0 &&this.Anim == 'Talk'){
       if(this.times == 200){this.currentAnim = this.anims.Talk;}
       this.times = this.times -1;
       }
     if(this.times == 0){
       this.currentAnim = this.anims.idle;
       this.times = -1;
       }
     this.parent();
     },
```

The animation remains in place for `200` frames; when done Talkie returns to his idle state.

`Talkie` needs to check if the player is in the vicinity so he can start talking. When the player is close, the `textballoon` entity is spawned and Talkie will not talk again. `ig.game.sortEntitiesDeferred()` resorts the entities in the game by its z-value; this way you are sure the balloon is shown on top. The following code is used for this purpose:

```
   check: function(other){
     if(this.talked == 0){
       this.Anim = 'Talk';
       this.talked = 1;
       ig.game.spawnEntity('EntityTextBalloon',this.pos.x - 10,this.pos.y
   - 70,null);
       ig.game.sortEntitiesDeferred();
     }
   }
```

Now that we have our Talkie code finished, try adding him to one of the levels and get near him. A balloon should pop up saying **Epicness awaits you!**

Talkie is right because we are almost at the end of our game.

Summing up the preceding content, the steps are as follows:

1. We now need a character capable of delivering the message to the player. We will call this character `Talkie`.
2. Open a new JavaScript file and save it as `Talkie.js`.
3. Set up the `Talkie.js` file with the standard ImpactJS module code.
4. Include `Talkie.js` in your `main` script.
5. Add an animation sheet, an animation sequence, a size, name, and several other properties for the Talkie.
6. Add a property `talked`, which keeps track of whether Talkie already talked or not. And a property `times`, which is the number of frames for which the Talkie needs to look as if he is actually talking. The time span for which the talk animation is shown is best to be equal to the life span of the speech balloon.
7. Adapt the `update` function to make the speech animation work.
8. Overwrite the `check` function and collision detection so that a `textballoon` entity is spawned when the player touches the Talkie if he has not yet talked before.

The final battle

Usually a game ends with a grand finale; a boss of great strength, you need to slay in order to reap everlasting fame!

Let's have a look at the final `Boss` entity:

```
ig.module('game.entities.Boss')
.requires('plugins.ai','game.entities.enemy')
.defines(function(){
  EntityBoss = EntityEnemy.extend({
    name: 'Boss',/* Let's call him the Boss*/
    health: 300, /* he has more health than an ordinary enemy*/
    speed:80, /* The default speed is higher than an enemy*/
    animSheet: new ig.AnimationSheet('media/enemyboss.png',32,48)
    /* different animation sheet for the Boss */
    receiveDamage: function(amount,from){
      /* override the default because we want an end screen (or 
animation) */
      /* the boss is stronger then everyone, so he doesn't get damaged 
that fast */
      amount = amount / 2;
```

```
        if(this.health - amount <= 0){
        //ig.system.setGame(GameEnd); /*we want an end screen (or
animation)*/
        }
      /* update the health status */
      this.health = this.health - amount;
      }
    })
  });
```

In this case, the `Boss` entity is nothing more than a strong enemy. There is no need to do an exact copy and paste of the `enemy` entity and adjust code separately on the elements they share. It is more efficient to extend the `enemy` class and only fill out the differences. Our boss has another name, more health, more speed, looks different, and takes less damage from a hit. In order to be able to build upon the original `enemy` entity, you need to include it in its `require` function. As the enemy is already built upon the ImpactJS `entity` class, there is no more need to include `impact.entity`.

In addition, we need to tell the `projectile` entity that it is also allowed to hit the `Boss` entity, as shown in the following code snippet:

```
check: function(other){
  if (other.name == "enemy" || other.name == "Boss"){
    other.receiveDamage(100,this);
    this.kill();
    this.parent();
    }
  }
```

In `projectile.js`, the `if` statement is adjusted to cope with our `Boss` entity. You might have noticed that the death of our enemy triggers the end of the game. We will look into that and the opening splash screen in *Chapter 5*, *Adding Some Advanced Features to Your Game*. You can add a `Boss` entity to the `endgame` level and fight him for glory!

Summing up the preceding content, the conclusions are as follows:

- A final boss is often the long anticipated antagonist that a player needs to defeat in order to finish a game or stage. He often has more health, does more damage, and thus is generally harder to defeat than a regular foe.
- We can base our boss's character on a regular enemy by extending the `enemy` class when creating the `boss` entity.

Summing up the preceding content, the steps are as follows:

1. Open a new JavaScript file and save it as `Boss.js`.
2. Set up the `Boss.js` file by extending the `enemy` class.
3. Include `Boss.js` in your `main` script.
4. Change all the properties that need to distinguish the boss from a lesser foe. This includes health, damage, speed, and even armor. Armor can be implemented as a damage reduction by overwriting the `receivedamage()` function.
5. Overwrite the `receivedamage()` function to make sure the end of the game is called when the boss dies. This GameEnd is explained in *Chapter 5, Adding Some Advanced Features to Your Game* so you can turn it off for now.
6. Adapt the `projectile` entity so it also damages the `Boss` entity and not just the `enemy` entity.

Summary

In this chapter we have been able to build our own top-down game from scratch. In order to do this, we have built levels with the ImpactJS Weltmeister and added a controllable character known as **player**. The game becomes more challenging by adding intelligent enemies and the weapons to defeat them. We were able to add some more depth to the game by introducing a friendly NPC. The final element was keeping score in order to provide the player with some feedback on how well he or she is doing.

4
Let's Build a Side Scroller Game

In this chapter we will build a very basic side scroller game using both ImpactJS and Box2D. Box2D is an open source C++ physics engine. With it, the forces of gravity and friction are simulated like you would see in a game of Angry Birds. Although not perfectly integrated, but given enough effort, Box2D can be used within an ImpactJS game. Just like the previous chapter, a game will be built from the ground up. The main difference will be in the use of a physics engine and the side scroller game setting.

In this chapter we will cover:

- Side scroller game
- Using Box2D with ImpactJS
- Building a side scroller level with the ImpactJS Weltmeister
- Introducing a playable character
- Adding some enemies to the side scroller
- Arming the player with bullets and bombs
- Making the enemy smarter with artificial intelligence
- Creating items which can be picked up by the player
- Keeping score and adding points every time an enemy dies
- Connecting two different side scroller levels
- Ending the game with a strong foe

The side scroller game setting

A side scroller videogame is a game which is viewed from a side angle and the player generally moves from left to right while playing through a level. The screen basically scrolls to the side, whether it is from the left to the right or any other direction, hence the name, side scroller. Well known side scroller games are 2D Mario, Sonic, Donkey Kong, the old Mega Man, Super Nintendo and Gameboy Metroid games, and the ancient but successful Double Dragon.

Most games of this type feature a long level through which the hero needs to find his way by battling or avoiding monsters and death traps. On reaching the end of the level, there is usually no way back except for replaying that particular level from the start. Metroid was a slight oddity in this respect, since it was one of the first side scrollers to feature an enormous world you could explore just like you would in a standard Role-playing game (RPG). Metroid set the stage for a new way of thinking about side scrollers; you needed to find your way in any possible direction through virtual miles of caves and you would occasionally find yourself back at the starting point. Castlevania is another example of a side scrolling adventure game, this time using a medieval setting.

Now that we have taken a look at what a side scroller is, let's get to building one with ImpactJS.

Preparing the game for Box2D

Before we can start in earnest, we need to make sure all files are correctly in position:

1. Make a copy from the original ImpactJS downloadable folder we prepared in *Chapter 1*, *Firing Up Your First Impact Game*. Alternatively, you can again download a fresh one and also put it in your XAMPP server's `htdocs` directory. Give your folder a name; let's be totally original and call it `chapter4`. Any other name will be fine too.
2. Download the physics demo from the ImpactJS website and go to its `plugins` folder. Here you should find the `Box2D` plugin. Create your own `plugins` folder and drop the `Box2D` extension there.
3. Test if everything works so far by visiting `localhost/chapter4` in your browser. The **it works!** message should once again be waiting there for you.
4. Additionally, we will need to change some Box2D core files. Box2D is not a product of ImpactJS but was invented for C++ based games before a JavaScript equivalent was developed. This JavaScript version was then integrated with ImpactJS by Dominic Szablewski (creator of ImpactJS). However, there are some flaws, one of them being faulty collision detection. Therefore, we will need to swap one of the original files with an adapted one that takes care of this problem. Take the `game.js` and `collision.js` scripts from the downloadable files for the `chapter4` folder and place them in your local `Box2D` folder. The `collision.js` script is with thanks to Abraham Walters who supplied the script.
5. Copy the `chapter4` folder's media files to your local `media` folder.
6. We need to make an adaption to the main script. Our game will no longer be an extension of the standard Impact game class.

   ```
   MyGame = ig.Game.extend({
   ```

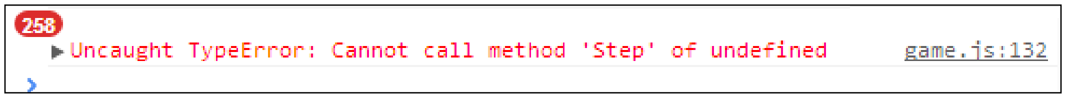
7. Instead it will be an extension of the modified Box2D version. So make sure to change the following piece of code:

   ```
   MyGame = ig.Box2DGame.extend({
   ```

8. We require the Box2D `game` file in order to work with this extension, so include it at the beginning of the `main.js` script.

   ```
   .requires(
     'impact.game',
     'impact.font',
     'plugins.box2d.game'
   )
   ```

9. Finally, in order to test whether everything is working fine, we will need to load in a level with a collision layer. This is because Box2D needs the collision layer to create its world environment and their boundaries. Without a level, you will come across an error, that looks like the following:



```
258
▶ Uncaught TypeError: Cannot call method 'Step' of undefined      game.js:132
```

10. To do this, copy the `testsetup.js` script from the `level` subfolder of the `chapter4` folder, and place it in your local `levels` folder. Add the level to your required files.

    ```
    'game.levels.testsetup'
    ```

11. Insert a `loadlevel()` function in the main script's `init()` method.

    ```
    init: function() {
        this.loadLevel( LevelTestsetup );
      },
    ```

12. Reload the game in your browser and you should see the **it works!** message. Now you have seen it, you can delete it from your code. It is inside the main script's `draw()` method.

    ```
      var x = ig.system.width/2,
        y = ig.system.height/2;
        this.font.draw( 'It Works!', x, y, ig.Font.ALIGN.CENTER
    );
    ```

Great! We should now have everything ready to go. The first thing we will do is build a small level to have a playground of our own.

Building a side scroller level

In order to build a level, we once again need to rely on the ImpactJS Weltmeister:

1. Open the Weltmeister in your browser `localhost/chapter4/Weltmeister.html`. We don't have any entities to play around with, so all that we will add for now is some graphics and a collision layer. This collision layer is especially important since the Box2D extension code will look for it and lack of it will crash the game. Suffice to say Box2D for ImpactJS is still in its infancy and minor bugs like this are to be expected.

2. Add a layer and name it `collision`; the Weltmeister will automatically recognize it as a collision layer.
3. Set its tilesize to `8` and the layer dimensions to `100 x 75`. We now have an 800 x 600 pixel canvas to play with.
4. Now draw a box around the edges so we have a closed environment from which no entity can escape. This will be very important when gravity comes into play. Without a solid ground you will get some unintended results for sure.
5. Now add a new layer and call it `background`. We will use a single picture as background for this level.
6. Select the `church.png` file from the `media` folder as tileset. Our picture is 800 x 600 pixels so it should fit exactly in the area we created with our collision layer. Set the tilesize to `100` and the layer dimensions to `8 x 6`. Paint the picture of the church on the canvas.
7. Save your level as `level1`.

Great, we now have a basic level of our own. It is pretty empty though and some extra obstacles would be nice. Just follow the next steps to add some obstacles:

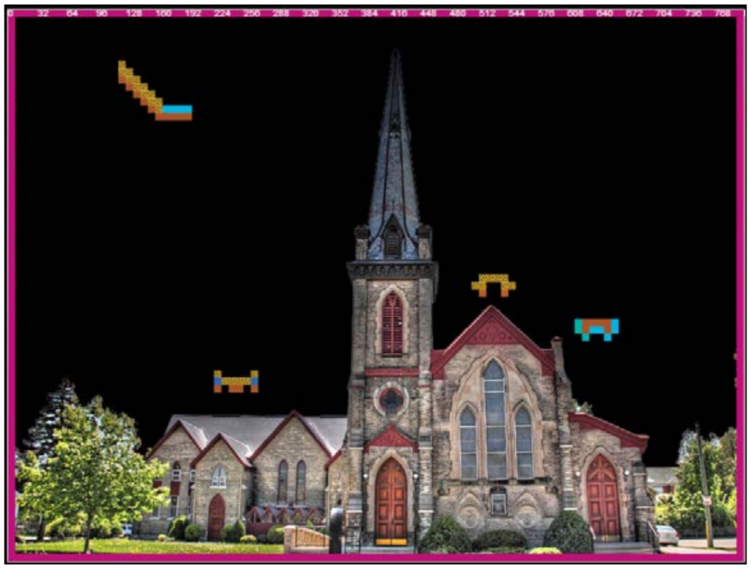
1. Add another layer called `platforms`.
2. Use the `tiles.png` file as tileset. They are simple in design but will do just fine as the building blocks for any platform you like to construct. Set the tilesize to `8` and dimensions to `100 x 75`, exactly like the collision layer.
3. Turn on the option **link with collision layer** before you start painting the platforms. This way, you don't need to trace the platforms with your collision layer afterwards. If you don't want every part of a platform to be solid, you can, of course, temporarily turn off the link, paint the tiles, and turn it back on; the link is not made retrospectively.
4. Add a few floating platforms to the level; follow your heart's desire on what they should look like.
5. Save your level when you feel the stage is set.
6. Add the level to the `require()` function of your `main.js` script.

```
.requires(
  'impact.game',
  'impact.font',
  'plugins.box2d.game',

  'game.levels.testsetup',
  'game.levels.level1'
)
```

7. Make sure that the level called `level1` is loaded at the beginning, instead of our `testsetup` level, by changing the `loadLevel()` function parameter.

```
    init: function() {
    // Initialize your game here; bind keys etc.
    this.loadLevel( LevelLevel1 );
  },
```



It is time to add a playable entity to the game so that we can discover the breathtaking level we have just created.

The playable character

Since we are working with Box2D, we will not use the standard ImpactJS entity but the adapted version. Especially the way an entity moves in a Box2D world is what makes all the difference. In standard ImpactJS, this is the very straightforward process of moving your character image a few pixels in a certain direction. However, Box2D works with forces; so, in order to move, you need to overcome gravity and even air friction. But let's start by setting up a basic entity:

1. Open a new JavaScript file and save it as `player.js` in the `entities` folder.
2. Add the basic Box2D entity code as follows:

```
ig.module(
  'game.entities.player'
)
```

```
    .requires(
      'plugins.box2d.entity'
    )
    .defines(function(){
      EntityPlayer = ig.Box2DEntity.extend({
      });
    });
```

3. As you can see, the term `entity` is an extension of the Box2D entity and, as such, requires the Box2D entity plugin file. Once again, make sure the naming conventions are respected or your player entity will not show up in Weltmeister.
4. Add the `'game.entities.player'` parameter to the `main.js` script.

If you were to visit your Weltmeister after making these modifications, you would find the player in your entity's layer. He doesn't represent much yet though; currently it is nothing more than an invisible square over which you don't have any control. Time to change his invisibility by adding an animation sheet.

```
EntityPlayer = ig.Box2DEntity.extend({
  size: {x: 16, y:24},
  name: 'player',
  animSheet: new ig.AnimationSheet( 'media/player.png', 16, 24 ),
  init: function( x, y, settings ) {
    this.parent( x, y, settings );
    this.addAnim( 'idle', 1, [0] );
    this.addAnim( 'fly', 0.07, [1,2] );
  }
});
```

With the previous block of code, we gave the player a size and a name; but more importantly, we added graphics. The animation sheet contains just two images, one for the player standing idle and the second one for when the player flies. This is not much but it will suffice for a simple game. A side scroller game has a considerable advantage when it comes to its need for graphics. In theory, you only need two images to represent a character; that is, one for the character standing still and another one for the character in motion. For a top-down game you would need at least six images to accomplish the same thing. This is because, in addition to a side-view, you need an image of the back and the front of the character. Consequently, if you would add an animation for the player firing a bullet, this would result in an extra drawing for the side scroller but three for the top-down game. It's clear how a side scroller game is better, if you have only limited resources to get your graphics.

It's just great that we can now add our player to the game and he is actually visible, but we don't have any control over him yet.

Adding player control is done at two places, namely, the main and the player script. In the main script, add controls to the game's `init()` method.

```
init: function() {
    // Bind keys
    ig.input.bind(ig.KEY.LEFT_ARROW, 'left' );
    ig.input.bind( ig.KEY.RIGHT_ARROW, 'right' );
    ig.input.bind( ig.KEY.X, fly);
//Load Level
    this.loadLevel( LevelLevel1 );
  },
```

In the player script, we need to change our `update()` function so the player can react to our input commands.

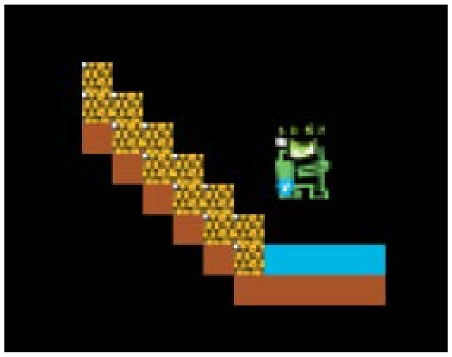
```
update: function() {
  // move left or right
  if( ig.input.state('left') ) {
    this.body.ApplyForce( new b2.Vec2(-20,0),
this.body.GetPosition() );
    this.flip = true;
  }
  else if( ig.input.state('right') ) {
    this.body.ApplyForce( new b2.Vec2(20,0),
this.body.GetPosition() );
    this.flip = false;
  }
  // jetpack
  if( ig.input.state('fly') ) {
    this.body.ApplyForce( new b2.Vec2(0,-60),
this.body.GetPosition() );
    this.currentAnim = this.anims.fly;
  }
  else {
    this.currentAnim = this.anims.idle;
  }
  this.currentAnim.flip.x = this.flip;
  this.parent();
}
```

In Box2D the entity has an extra property, a body. In order to move the body, we need to exert a force upon it. This is exactly what happens when we use the body's `ApplyForce()` method. We apply a force in a certain direction, so we actually use a vector. The use of vectors is what Box2D is all about. As long as we keep the right, left, or fly button pressed, the force is applied. However, when released, the entity doesn't stop immediately. No further force is applied but it takes a certain amount of time for the effects of the applied force to wear out; this is a big difference from the velocity we used in the previous chapters.

If you add the player to the level, make sure he is somewhere on top of a platform in the top-left corner. The top-left corner is what will be visible by default and we don't have an adaptable viewport to follow our player around just yet. To be precise, he doesn't really need a platform to stand on just yet, since we have no gravity in our world. Let's do something about that. Add the gravity property to your game in the `main.js` script as follows:

```
MyGame = ig.Box2DGame.extend({
  gravity: 100,
```

Let's take our player for a test-flight shall we?



You might have noticed that even though he flies rather smoothly, any solid object our jetpack frog encounters will make him go into a spin. Probably you don't actually want this to happen. This is especially annoying since he can end up with his head facing down, at which point his jetpack flame is facing upwards. Now with the jetpack flame facing upwards, it doesn't make a lot of sense if activating the jetpack still results in an upwards thrust. Therefore, we need to do something about his stability. This can be done by fixing the body on the horizontal axis. Add the following code to the frog's `update()` function:

```
this.body.SetXForm(this.body.GetPosition(), 0);
```

Now the player's body is fixed at an angle of 0 degrees towards the x axis. Try changing it to 45; you now have a crazy frog, always flying with his body tilted to the right, even when facing to the left.

We have a flying and steady frog now. Just too bad we can't see him once we move a bit too far to the right or gravity takes us to the bottom of the level. It's definitely time to introduce a following camera. To do this we need to make a modification to our game's `update()` function as follows:

```
update: function() {
    this.parent();
    var player = this.getEntitiesByType( EntityPlayer )[0];
    if( player ) {
      this.screen.x = player.pos.x - ig.system.width/2;
      this.screen.y = player.pos.y - ig.system.height/2;
    }
},
```

The player is put in a local variable and its location is checked once per frame to update the screen's position. Because we subtract half of the viewport's size from the players position, our player is kept neatly in the center of the screen. Leaving out the subtraction will keep the player in the top-left corner of the screen.

Save all the modifications and fly around the level you created; enjoy the peace and silence as much as you can because soon hostiles will stir up the place.

Let's quickly recap what we have covered about the Box2D entity and how we can make a playable character with it. A Box2D entity is different from an ImpactJS entity, such that Box2D makes use of vectors to move around. A vector is a combination of direction and force:

- Open a new JavaScript file and save it as `player.js`.
- Insert the standard Box2D entity extension code.
- Include the player entity in the main script.
- Add animation to the player. Also make use of the `flip` property, which flips your image over a vertical axis and cuts the needed character graphics in half for a side scroller game.
- Add player controls which grant the ability to move left, right, and upwards. Notice how force is applied on the body in order for it to move. Once the input button is released and no more force is applied, the entity will proceed on course and completely stop, once the force is totally dissipated or he hits a solid wall.

- Introduce gravity as a property of the game. As gravity is a constant downward force, it will drag everything down to the first solid object it encounters, unless a counter-force is provided. For our flying frog, his jetpack is his counter-force against gravity.
- Our frog doesn't know how to fly a steady course just yet. Fix him to the horizontal axis so he doesn't go spinning every time he hits a solid object.
- Finally, we need a camera to keep track of where we are going. Incorporate the automatically following camera in the game's `update()` function.

Adding a minor foe

We need some opposition, something we can shoot out of the sky once we have the weapons to do so. Therefore, let's introduce some more frogs! Hostile ones this time:

1. Open a new file and save it as `enemy.js`.
2. Insert the following code into the file. It is the minimum code necessary to get a representation of our enemy in the Weltmeister. As such, it already includes the animation sheet.

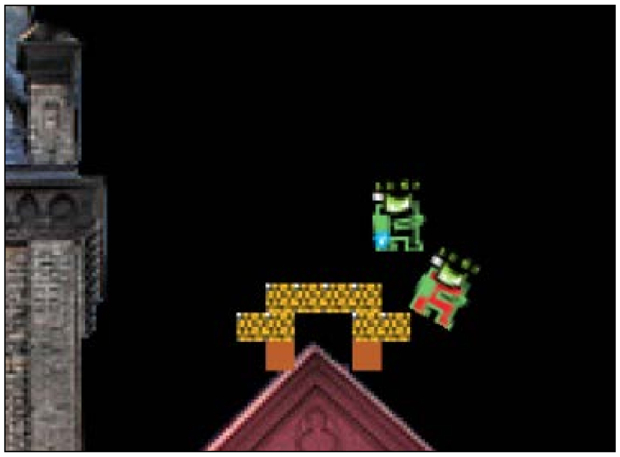
```
ig.module(
  'game.entities.enemy'
)
.requires(
  'plugins.box2d.entity'
)
.defines(function(){
EntityEnemy = ig.Box2DEntity.extend({
size: {x: 16, y:24},
name: 'enemy',
animSheet: new ig.AnimationSheet( 'media/enemy.png', 16, 24
),
init: function( x, y, settings ) {
  this.parent( x, y, settings );
  // Add the animations
  this.addAnim( 'idle', 1, [0] );
  this.addAnim( 'fly', 0.07, [1,2] );
  }
})
});
```

3. Require the enemy entity in our `main.js` script.

```
'game.entities.enemy'
```

4. Add an enemy to the level with the Weltmeister.

Since our enemy is currently pretty defenseless, we could just knock him off a platform as well.



In normal ImpactJS code, we would have to set collision variables for this to happen, otherwise, the player and the enemy frog would just go straight through each other. In Box2D, this is not necessary as collision is automatically assumed and a force is applied on every movable object we hit with our flying frog.

Since we have gravity working, a great alternative for placing our enemies at a certain location is spawning them in at the top of the level. Add the `spawnEntity()` function in the game's `init()` function. An enemy will spawn there and gravity will drag it down to the bottom.

```
this.loadLevel( LevelLevel1 );
    this.spawnEntity('EntityEnemy',300,30,null);
```

Make sure the `spawnEntity()` function is used after the level loads or an error occurs. Once the enemy has intelligence of its own, spawning enemies at the top of the level will make more sense. They will drop down, either to the very bottom or until they reach a platform, where they will wait for the player and attack it.

We will turn our red frog into a really annoying creature once we provide him with some basic artificial intelligence. However, let's first prepare ourselves by adding some weapons to the game.

Let's briefly recap how we created our enemy:

- Open a new JavaScript file and save it as `enemy.js`
- Insert the standard Box2D entity extension, attach an animation sheet, and add animation sequences

- Include the enemy entity in the main script
- Add an enemy to the level with the Weltmeister and the `spawnentity()` method

Introducing formidable weapons

Weapons are great, especially if they are subjected to gravity or if they can apply some force to other entities. We will have a look at two types of weapons here, namely, the projectile and the bomb.

Shooting a projectile

The projectile will be our main weapon against the rival frogs, so let's start by setting up the basics:

1. Open a new JavaScript file and save it as `projectile.js` in the `entities` folder.
2. Add the basic Box2D entity code with animation sheet and sequences as shown in the following code snippet:

```
ig.module(
  'game.entities.projectile'
)
.requires(
  'plugins.box2d.entity'
)
.defines(function(){
  EntityProjectile = ig.Box2DEntity.extend({
  size: {x: 8, y: 4},
  lifetime:60,
  name: 'projectile',
  animSheet: new ig.AnimationSheet( 'media/projectile.png', 8, 4
),
  init: function( x, y, settings ) {
    this.parent( x, y, settings );
    this.addAnim( 'idle', 1, [0] );
  }
});
});
```

3. Apart from a name, size, and the necessary elements to perform the animation, we have already included a property called `lifetime`. Every projectile starts with a `lifetime` of `60`. We will make this drop and kill off the bullet when it reaches `0`. This way we don't get an overload of entities in a single game. Every entity needs its own calculations and having too many of them on screen at once might significantly drop the game's performance. Keeping track of this performance can be done using the ImpactJS debugger, which can be turned on by including the `'impact.debug.debug'` command, in the main script.
4. Add the `game.entities.projectile` script to the `main.js` script.

We can now add projectiles to the game by using Weltmeister if we want. However, adding them manually is not of much use to us. Let's adapt the player's code instead so our frog can spawn the projectiles. First, bind the `'shoot'` state to a key in the main script.

```
ig.input.bind(ig.KEY.C, 'shoot' );
```

Then add the following code to the player's `update()` function.

```
if(ig.input.pressed('shoot') ) {
  var x = this.pos.x + (this.flip ? -0 : 6);
  var y = this.pos.y + 6;
  ig.game.spawnEntity( EntityProjectile, x, y, {flip:this.flip} );
}
```

Spawning the projectile needs to be done at a certain location and it has to point in a specific direction, either left or right. We arbitrarily set the y coordinate of the spawnpoint as `6` pixels lower than our player's position; we can make this 10, 20, or 200 pixels as well. Though, in the case of the last option, it would look as if the bullet was spawned way below the player, which would be rather unusual. However, let's not forget that the player's position is always at the top-left corner of its image. Given that the height of our frog is 24 pixels, it will appear as if the bullet is spawned from the mouth, which is pretty cool for a frog. The x coordinate is another matter. If the frog is facing left, we don't adjust the spawn coordinate; if he's facing right, we adjust it by 6 pixels. The information about whether the player is flipped is not only used to adjust the spawn coordinate. It is also transferred to the projectile itself as an optional input parameter. Here it will be used to determine what side it should face and fly. While firing the bullet you might notice the frog being kicked back a little, not unlike the recoil of a gun. This is because the frog initially occupies a space the bullet comes to occupy, when spawned. If you want to avoid this cool effect, all you need to do is spawn the bullet a bit farther away from the frog. If you were to load the game at this point, you would notice your bullet spawns but doesn't fly away. That is because we didn't tell the bullet to do so whenever it comes into existence.

Adding the following two lines of code to the projectile's `init()` function will rectify this situation.

```
this.currentAnim.flip.x = settings.flip;
var velocity = (settings.flip ? -10 : 10);
this.body.ApplyImpulse( new b2.Vec2(velocity,0),
this.body.GetPosition() );
```

Upon spawning the projectile we now apply an impulse instead of a force. There is a significant difference between the `ApplyImpulse()` and `ApplyForce()` functions. While the `ApplyForce()` function applies a constant force on the body, the `ApplyImpulse()` function only applies it once, but abruptly. You could compare it to pushing a rock versus running up to it and slamming into it with your full force and all the momentum you gathered. A bullet in real life works the same way as we try to simulate it here; it is flung away by a small explosion and never pushed again thereafter. The local variable, `var.velocity`, is there to adapt the bullet's direction in the same way the animation depends on the value of the `settings.flip` parameter. If the value of the `flip` property is false, the bullet will be facing right and flying to the right. If the value of the `flip` property is true, the animation is flipped, making it face left. Because the velocity then takes a negative number, the bullet also flies to the left.

We could still adapt our impulse on the y axis which is currently set at `0`. Putting a negative number will make our frog shoot upwards like an anti-air gun. A positive number will make him shoot downwards like a bomber. Try playing around with this to see the effects.

Our projectiles are still lingering around, cluttering the screen, because we haven't put our `lifetime` property to good use yet.

Let's modify the `update()` function in order to limit the lifespan of our bullets.

```
update: function(){
  this.lifetime -=1;
  if(this.lifetime< 0){this.kill();}
  this.parent();
}
```

Every time the game goes through the update loop, which is once per frame, the remaining lifetime of the projectile is shortened by 1. Given a total lifetime value of 60 in a 60 frames per second game, the bullet has 1 second to live after being spawned.

We can shoot at our enemies with it and actually push them away with the force of the bullet, but we don't really damage them yet. For this to happen we need to check if we hit an enemy.

```
check: function(other){
  other.receiveDamage(10);
  this.kill();
}
```

Adding this modified `check()` function, which will make the projectile deal damage before killing itself, is not enough. Even though collision is automatically handled by Box2D, the parameters necessary for a `check()` function to work are not. We need to do a few more things:

1. Tell the enemy it is a type B entity by adding the `TYPE` property as follows:

   ```
   type: ig.Entity.TYPE.B,
   ```

2. Make the projectile check for collision with type B entities using the `checkAgainst` property.

   ```
   checkAgainst: ig.Entity.TYPE.B,
   ```

3. Now save and reload the game. You can now kill those nasty red frogs.

Try making your player a type B entity. Your bullets will now kill you. This is because we made them spawn in a space already occupied by our frog. As we have seen before, this is also the reason why we have this recoil effect upon firing a bullet. However, this time it's more than recoil; it can actually kill the player. So we better not make our player a type B entity or we should spawn our bullet a bit farther away and lose the recoil effect. It's great to have something to defend ourselves with, even if the other frogs don't pose much of a threat just yet. Before bringing them to life, we are shortly going to have a look at something more explosive, a bomb.

Before moving on to the bomb, let's again have a quick look at how we introduced our main weapon, the bullet:

- We need guns, lots of them.
- Open a new JavaScript file and save it as `projectile.js`.
- Insert the standard Box2D entity extension, attach an animation sheet, and add animation sequences. Also add a `lifetime` property, which will keep track of how much longer the bullet should stay in the game.
- Include the projectile entity in the main script.
- Add a key bind to the shoot input state in the main script.
- Make our flying frog spawn a projectile when the player clicks on the shoot button.

- Add an impulse to the bullet so it can actually fly instead of just dropping to the ground.
- Check how long the bullet is in the air and kill it off if it exceeds its preset lifespan.
- Let the bullet check for enemies. If it encounters an enemy, it should deal damage and kill itself.
- Try making the bullet kill the player but don't keep it that way.

Building an actual bomb

The basics for making a bomb are the same as for the projectile, indeed, they are the same as for creating any entity:

1. Open a new JavaScript file and save it as `bomb.js` in the `entities` folder
2. Add the basic Box2D entity code with the animation sheet and sequences as follows:

```
ig.module(
  'game.entities.bomb'
)
.requires(
  'plugins.box2d.entity'
)
.defines(function(){
EntityBomb = ig.Box2DEntity.extend({
  size: {x: 24, y: 10},
  type: ig.Entity.TYPE.A,
  checkAgainst: ig.Entity.TYPE.B,
  animSheet: new ig.AnimationSheet( 'media/bomb.png', 24, 10 ),
  lifespan: 100,
  init: function( x, y, settings ) {
    this.parent( x, y, settings );
    // Add the animations
    this.addAnim( 'idle', 1, [0] );
    this.currentAnim = this.anims.idle;
  }
});
});
```

3. This time we already gave our bomb a type and a type to check against for inflicting damage
4. Put the `game.entities.bomb` parameter as a required entity into the `main.js` script

We now have a bomb we can put anywhere in the level we want to. We can add some bombs close to the ceiling of our level so they would fall down at level load. That would be great, given there would be an actual explosion. We are going to introduce that explosion as a separate method that only our bomb can use.

```
explosion:
function(minblastzone,maxblastzone,blastdamage,blastforcex,
blastforcey){
  varEnemyList= ig.copy(ig.game.entities);
  var i = 0;
  //check every entity
  while(typeofEnemyList[i] != 'undefined'){
    Enemy = EnemyList[i];
    //calculate distance to entity
    distance = Math.sqrt((this.pos.x - Enemy.pos.x)*(this.pos.x -
Enemy.pos.x) + (this.pos.y - Enemy.pos.y)*(this.pos.y -
Enemy.pos.y));
    //adjust blastdirection depending on entity position
    if(this.pos.x - Enemy.pos.x< 0){adjustedblastforcex =
blastforcex}
    else{adjustedblastforcex = - blastforcex}
    if(this.pos.y - Enemy.pos.y< 0){adjustedblastforcey = blastforcey}
    else{adjustedblastforcey = - blastforcey}
    //if within blastzone: blow up the target
    if(minblastzone< distance && distance <maxblastzone){
      Enemy.body.ApplyImpulse(new
b2.Vec2(adjustedblastforcex,adjustedblastforcey),
this.body.GetPosition());
      Enemy.receiveDamage(blastdamage,this);}
      i++;
    }
}
```

Just like the `init()`, `update()`, and `check()` methods, we now insert the `explosion()` method into the bomb entity so it will be able to use it henceforth. The `explosion()` method takes in five parameters:

1. **Minimum blast-zone**: If an entity is closer than this distance, he will not be hit. This doesn't make a lot of sense for bombs, except that it allows you to use several blasts in one bomb. This in turn makes it possible to have higher damage when the target is closer to the bomb and lower damage when he is farther away.
2. **Maximum blast-zone**: Everything farther away than the maximum blast-zone will not be affected by the explosion.

3. **Blast damage**: This is the damage an entity will receive if it is within the blast-zone.
4. **Blastforcex**: This is the impulse on the x axis that will be applied to the affected entity. It will decide how far the target will fly to the right or left.
5. **Blastforcey**: This is the impulse on the y axis that will be applied to the affected entity. It would determine how high the target will fly. Obviously, if the target is below the bomb when it explodes, it will push the target downwards, not upwards.

The `explosion()` method works in the following manner. All entities are copied to a local variable. The entities are then checked one after the other to see what their distance from the bomb is. The distance is calculated as Euclidean distance here. While calculating the Euclidean or ordinary distance, you apply the Pythagorean formula. This formula states that, the length of any side of a triangle with a 90 degree angle can be calculated if the length of the other two sides is known. The formula is $a^2 + b^2 = c^2$, with c being the longest side of the triangle. Depending on whether the hapless target is situated to the right or left, above or below the bomb, the direction of the force is adjusted. Finally, the function checks if the distance is within the limits of the blast-zone. If this is the case, a damage and an impulse are applied on the target. At this point either the entity dies or goes flying through the air; either way it's not good news.

Simply adding this `explosion()` method will do no good until we actually use it. Therefore, we need to modify our `update()` method so we can blow up our bomb at the end of its lifespan.

```
update: function(){
  //projectiles disappear after 100 frames
  this.lifespan -= 1;
  if(this.lifespan< 0){
    this.explosion(0,40,70,200,100);
    this.explosion(40,200,20,100,50);
    this.kill();
  }
  this.parent();
},
```

The lifespan part works exactly like it did in the projectile. However, in this case we don't merely call the `kill()` function, we use our newly developed explosion twice. We could just call the function once and put the blast-range value between 0 and 200. As mentioned earlier, the advantage we have now is the division between high damage and pressure close to the bomb, and low damage and pressure farther away. Technically, we can use as many explosions as we like; each one of them requires calculation time. Though, it's up to you on how much division you want.

Before you actually get to testing this explosion in your game, make sure to attribute health to all your entities. Whether they will survive the damage of the blast or not will depend on whether you have given them enough health. Since the default value is set at 10, they will not fly away, but die instantly if the previous numbers are used. So let's give our player and enemies a health value of 100 by adding this property before their respective `init()` functions.

```
health: 100
```

As a final touch we can make the bomb explode when it comes into contact with one of our hostile frogs.

```
check: function(other){
    other.receiveDamage(30);
    this.explosion(0,40,70,200,100);
    this.explosion(40,200,20,100,50);
    this.kill();
}
```

We already made sure the bomb checks for contact with entities of type B by setting the `checkAgainst` property. The direct damage of getting this piece of metal on the face is set at `30`. This is followed by the explosion itself, which will deal damage worth 70 points because the enemy is so close. The second blast wave affects everything farther away before the bomb finally kills itself.

Now we have a bomb that can be put anywhere in the level and will do just fine. However, it would be even better if our player could spawn one himself. In the following steps, we simply repeat what we did with the projectile to make the player spawn a bomb by himself:

1. Assign a keyboard button to your bomb input state as shown in the following line of code:

   ```
   ig.input.bind(ig.KEY.V, 'bomb');
   ```

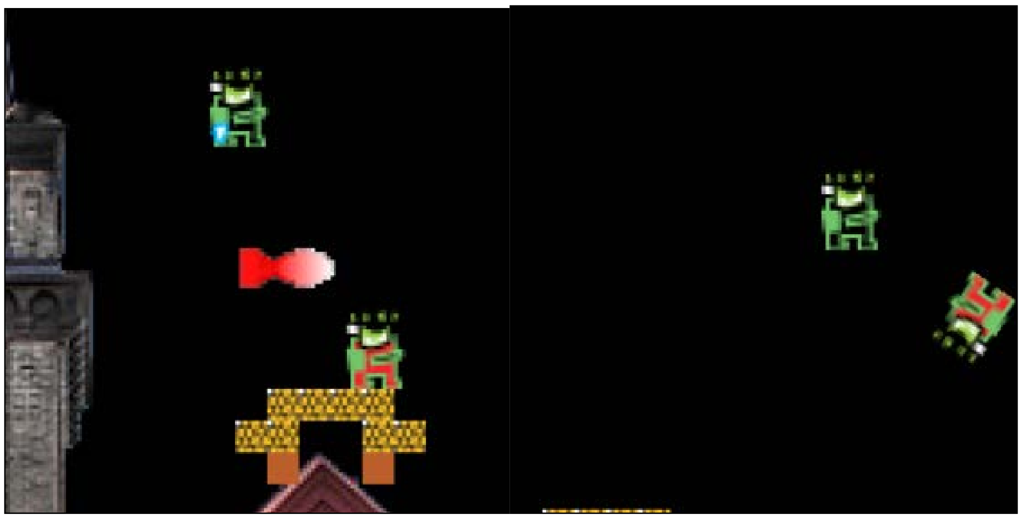
2. Change the player's `update()` function so the player can now spawn bombs with the help of the following code:

   ```
   if (ig.input.pressed('bomb')){
     var x = this.pos.x + (this.flip ? 0 : 8 );
     var y = this.pos.y + 25;
     ig.game.spawnEntity(EntityBomb,x,y, {flip:this.flip});
   }
   ```

3. The spawn coordinates defined here are different from what we did with the projectile. The `y` coordinate is very important; it is set at `25` because our flying frog has a height of `24` pixels. This way the bomb is always spawned just beneath the flying frog.
4. Add the following line of code to the bomb's `init()` function so it takes the `flip` parameter to know which side to face when spawned.

```
this.currentAnim.flip.x = settings.flip;
```

5. Save, reload, and bomb the hell out of those red frogs! Watch out though, bombs can kill you too.



The bomb is our greatest weapon; let's have a quick recap on how we built it:

- Open a new JavaScript file and save it as `bomb.js`.
- Insert the standard Box2D entity extension, attach an animation sheet, and add animation sequences. Add a lifespan property that will keep track of how much time is left until the bomb explodes, if not detonated prematurely by touching an enemy.
- Include the bomb entity in the main script.
- Add a bomb to the level.
- Introduce the `explosion()` method; it is a custom function which simulates the damage and force effects of an explosion.
- Change the `update()` function so it will make the bomb explode when its time is up.

- Use the `check()` function to detect collision with an enemy and detonate immediately.
- Assign a keyboard shortcut to fire bombs.
- Adjust the player's `update()` function so a bomb is spawned when the player commands it to do so.
- Make the bomb flip to the direction the player is looking at.
- Mess around and have fun blowing frogs to bits!

Artificial intelligence

It is time to make our red frogs a bit smarter so they stand at least a tiny chance against our newly developed weapons arsenal. In *Chapter 3, Let's Build a Role Playing Game*, we did this completely by the book, by separating decisions from behavior. We made a separate Artificial Intelligence (AI) file for decision making, and—as always—the actual behavior is in the entity's `update()` function.

This time around, we will keep it very simple and put all AI directly in the enemy's `update()` method. This will demonstrate that even a simple AI can seem to act pretty smart.

Let's modify our enemy's `update()` function with the following code:

```
update: function(){
  var players = ig.game.getEntitiesByType('EntityPlayer');
  var player = players[0];
  // both distance on x axis and y axis are calculated
  var distanceX = this.pos.x - player.pos.x;
  var sign = Math.abs(distanceX)/distanceX;
  var distanceY = this.pos.y - player.pos.y;
  //try to move without flying, fly if necessary
  var col = ig.game.collisionMap.trace( this.pos.x, this.pos.y,
player.pos.x, player.pos.y,16,8 );
  if (Math.abs(distanceX) < 110){
    var fY = distanceY> 0 ? -50: 0;
    this.body.ApplyForce( new b2.Vec2(sign * -20,fY),
this.body.GetPosition() );
    if(distanceY>= 0){this.currentAnim = this.anims.fly;}
    else{this.currentAnim = this.anims.idle;}
  }
  this.body.SetXForm(this.body.GetPosition(), 0);
  if (distanceX> 0){this.currentAnim.flip.x = true;}
  else{this.currentAnim.flip.x = false;}
  this.parent();
  }
```

Inserting this function into your enemy entity will make him try to catch the player. But how does it work? First, the player entity is saved in the function's local variable called `player`. Both the horizontal distance and the vertical distance between the enemy and the player are calculated. The `sign` variable is used to determine whether the frog should fly to the left or the right. He will always fly upwards; if he needs to come down because the player is below him, he will just let gravity do its work. While flying, the fly animation is active, otherwise the idle animation is used, even while moving horizontally.

The frog's body is fixed to the x axis to prevent him from spinning, as is the case with the player. Finally the animation is flipped to the left or right, depending on where the player is relative to the enemy.

We now have a frog that will follow us around if we are close enough to him. Now we need him to do some damage to the player:

1. Make sure the type of the enemy and the type it needs to check for are filled out as B and A respectively. Also introduce a new enemy property called `cooldowncounter`, as shown in the following code:

   ```
   type: ig.Entity.TYPE.B,
   checkAgainst: ig.Entity.TYPE.A,
   cooldowncounter: 0,
   ```

2. The `cooldowncounter` property will keep track of how many frames have passed since the last time the frog was able to deal damage.
3. The `cooldowncounter` property must count, therefore add it to the `update()` function:

   ```
   this.cooldowncounter ++;
   ```

4. Extend the `check()` function so it checks whether enough frames have passed since the last attack and allow the frog to attack if this is the case, as shown in the following code:

   ```
   check: function(other){
     if (this.cooldowncounter> 60){
       other.receiveDamage(10,this);
       this.cooldowncounter = 0;
     }
   }
   ```

The frog will now be able to use its nasty melee attacks upon the player. The attack, whatever done by the frogs at a close range, will lower the player's health by a value of 10 every time it hits the player. The player will certainly need to avoid these nasty creatures now if he doesn't want to lose health quickly. We will need to give the player something extra that allows him to survive this carnage.

AI is what makes the enemy worth fighting against. Unlike what we mentioned in *Chapter 3, Let's build a Role Playing Game*, it does not always need to become a complicated matter though. Let's have a quick look at how we implemented AI for the side scroller:

- Change the `update()` function so the enemy can now fly towards the player. This new `update()` function is the AI of the enemy frog. As opposed to *Chapter 3, Let's Build a Role Playing Game*, both decision and behavior are wrapped in a single piece of code this time around.
- Introduce a cool down counter that keeps track of the number of frames since the last attack happened. Also make sure the enemy entity is of type B and checks whether it touches type A entities. The player should be a type A entity.
- Make the `cooldown` property add a value of 1 for every frame that passes, by adding it to the modified `update()` function.
- Incorporate the attack in the `check()` function so the frog becomes a force to be reckoned with.

Pickup items

Our tiny flying frog can now officially be killed by those nasty red frogs. This is not good news for him and it is up to us to provide a way to replenish the lost health. This is done by the use of pickup items, that is, entities which will disappear upon touching the player, but grant beneficial effects in the process.

Before we add an actual pickup item, which will come in the form of a health replenishing crate, let's first add a normal crate to the game.

Adding a normal crate

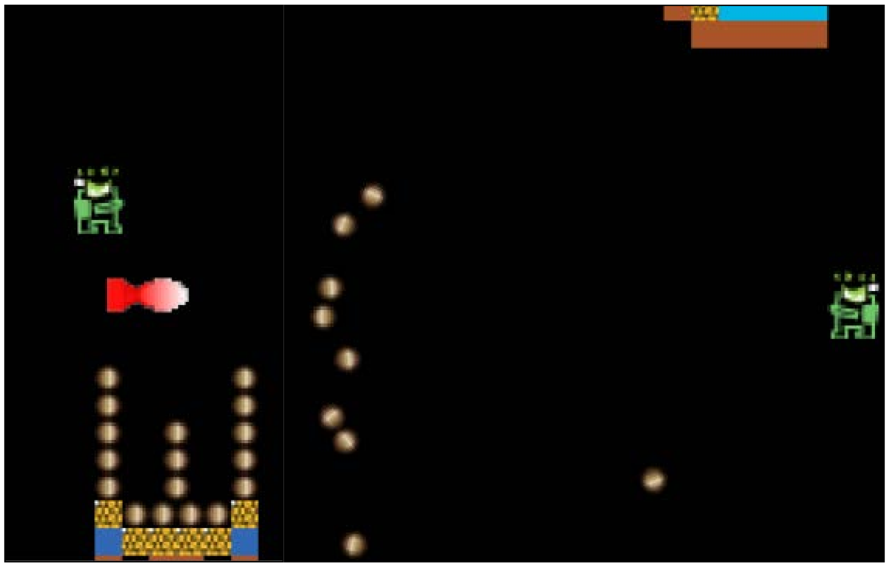
Our crate will function as a prototype to all types of crates we could invent. To create the crate, perform the following steps:

1. Open a new file and save it as `crate.js`.
2. Add the crate code to the file.

```
ig.module(
  'game.entities.crate'
)
.requires(
  'plugins.box2d.entity'
)
.defines(function(){
```

```
EntityCrate = ig.Box2DEntity.extend({
  size: {x: 8, y: 8},
  health: 2000,
  name: 'crate',
  type: ig.Entity.TYPE.B,
  checkAgainst: ig.Entity.TYPE.A,
  animSheet: new ig.AnimationSheet( 'media/crate.png', 8, 8
),
  init: function( x, y, settings ) {
    this.addAnim( 'idle', 1, [0] );
    this.parent( x, y, settings );
  }
});
});
```

3. This code is really straightforward since the crate is nothing but a lifeless object. A tough lifeless object though, since it has a health value of `2000`. By giving the crate so much health, it is able to survive multiple bomb explosions.
4. Save the file and add some to your game with the Weltmeister. Certainly try stacking a few of them before unleashing an explosion in their midst.



Now we have the standard crate; making a healing crate is only a few steps away, since we will build it upon our normal crate.

Before taking a look at our healing crate, let's quickly see how we made the normal crate:

- Create a new file and save it as `crate.js`
- Implement a standard Box2D entity code
- Save and add some crates to the game using the Weltmeister

Implementing a healing crate

Now we have our basic prototype crate, we only need to build upon it in order to create the health crate. To build the health crate perform the following steps:

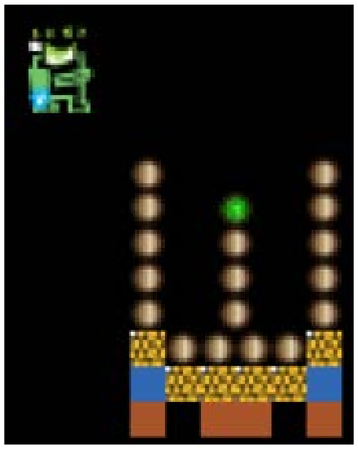
1. Open a new file and save it as `crate.js`.
2. Add the `healthcrate` specific code to it. The health crate is an extension of the normal crate, not a Box2D entity; therefore, we only need to point out where the health crate differs from the normal one:

```
ig.module(
  'game.entities.healthcrate'
)
.requires(
  'game.entities.crate'
)
.defines(function(){
EntityHealthcrate = EntityCrate.extend({
  name: 'healthcrate',
  animSheet: new ig.AnimationSheet( 'media/healthcrate.png', 8, 8
),
  check: function(other){
    if(other.name == 'player'){
      other.health =  other.health + 100;
      this.kill();
    }
  }
})
});
```

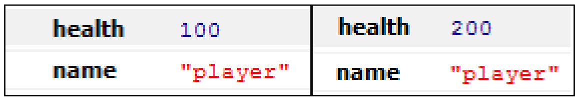
3. It has another name and animation sheet. In addition it will heal the player and destroy itself after healing the player.
4. Add the crate to the main script using the following code, so your game knows it's there.

```
'game.entities.healthcrate'
```

5. Save and add a crate to the game to check out its effects.



This crate heals the player by providing him a **health** value of **100**, as shown in the following screenshot. Thus, the player will always end up with more health than he started the game with. This is just a choice; you could always change this by implementing a health cap so as to make sure healing can't make the player stronger than he initially was.



health	100	health	200
name	"player"	name	"player"

Remember you can always open Firefox with the Firebug add-on and look for the player properties in **Document Object Model** (**DOM**). Before picking up the crate, our player had a health value of 100 and it rose to a value of 200 after he picked up the crate.

The healing crate is somewhat more sophisticated than the normal one. Let's once again have a look at the steps we took to make a healing crate:

- Create a new file and save it as `healthcrate.js`.
- Extend the previously built crate instead of a Box2D entity. Only add the parameters on which the health crate will differ from the original one. This includes a `check()` function to see if it has been touched by the player.
- Save and add a crate to the game using the Weltmeister.
- Check in the DOM if your health crate has actually increased your player's health.

Keeping score

Keeping track of a score within the game is a rather straightforward thing. In order to implement a system where a score is kept and increased every time an enemy is killed, we need three things:

1. We need a variable that keeps track and is within the boundaries of the game itself, but can be regarded as some sort of overhead variable.

   ```
   .defines(function(){
   GameInfo = new function(){
     this.score = 0;
   },

   MyGame = ig.Box2DGame.extend({
   ```

2. This is really important because, as we will see in *Chapter 5*, *Adding Some Advanced Features to Your Game*, the start and end screens are in effect different games that are being loaded. When a new game is loaded into memory, the old one is discarded and so are all of its variables. That is why we need a variable that exists outside of the game.
3. This function is used to increase the score by a certain number of points. This one is allowed to be a method of the game itself. Just insert this in the main script underneath the other main functions of the `MyGame` file.

   ```
   increaseScore: function(points){
     //increase score by certain amount of points
     GameInfo.score +=points;
   }
   ```

4. We overwrite the enemy `kill()` function as shown, so the frog does not only die but also supplies us with some extra points.

   ```
     kill: function(){
       ig.game.increaseScore(100);
       this.parent();
     }
   ```

From now on, every time a red frog dies, we get an extra 100 points and these are kept safely in a variable that is not erased as long as we do not refresh the page. We can then later on use this variable to show up at the end of a game to provide our gamer with some feedback of how well or poorly he did.

Keeping the score is a very important component for almost any game. It is a way to challenge the player to replay your game and do better at it. It is not too difficult to implement either; let's see what we did:

- Create a variable outside of the current game and call the variable `score`
- Add a function to the game that can directly manipulate our `score` variable
- Call the function when an enemy dies, to add points to the overall player score

Transitioning from one level to another

In order to have a map transitioning, you will first need a second level. You can make one yourself or copy the one from the downloadable files for this chapter. You will also need the trigger end `levelchange` entity. Copy both entities to the `entities` folder, and the level called `level 2` to the `levels` folder on your local computer. Alternatively, you can devise a second level yourself and use the trigger entity, which comes with your Impact license. The trigger entity is not part of the actual engine; it can be found in downloadable examples on the ImpactJS website.

In the `levelchange` entity, we will make the following code change:

```
ig.module(
  'game.entities.levelchange'
)
.requires(
  'impact.entity'
)
.defines(function(){
EntityLevelchange = ig.Entity.extend({
  _wmDrawBox: true,
  _wmBoxColor: 'rgba(0, 0, 255, 0.7)',
  _wmScalable: true,
  size: {x: 8, y: 8},
  level: null,
  triggeredBy: function( entity, trigger ) {
    if(this.level) {
      varlevelName = this.level.replace
(/^(Level)?(\w)(\w*)/, function( m, l, a, b ) {
        return a.toUpperCase() + b;
        });
      var oldplayer = ig.game.getEntitiesByType( EntityPlayer )[0];
      ig.game.loadLevel( ig.global['Level'+levelName] );
      var newplayer = ig.game.getEntitiesByType( EntityPlayer )[0];
```

```
          newplayer = oldplayer;
        }
      },
      update: function(){}
    });
    });
```

As you might notice, it is different from the one we used in the RPG, in two main aspects as follows:

- It does not take into account the use of spawnpoints. Using actual spawnpoints is not really necessary for most side scrollers. This is because once a level is completed, you can only go back to it by playing it all over again. Therefore, we do not need multiple spawnpoints per level, only one spawnpoint is required. If we only need one spawnpoint, however, it is way easier not to use the Void entity as we did in the previous chapters. Instead, we simply put the player entity on a certain location within the level and the level will thus always start there.
- The second change to the `levelchange` entity is the backup we make of the player entity. Before loading the level, we copy the player entity to a local variable called `oldplayer`. Once the game is loaded, a new playable character is created; this is the one we manually added to the level, `level 2` in the Weltmeister. We then assign this new player to another local variable called `newplayer`. By overwriting `newplayer` with `oldplayer`, we can keep playing with the old frog. This can be important if the player is allowed to keep the previously acquired supplementary weapons or health.

All we need to do now is correctly set up a `trigger` and a `levelchange` entity in `level 1` and we have a decent level transitioning. This should be done as follows:

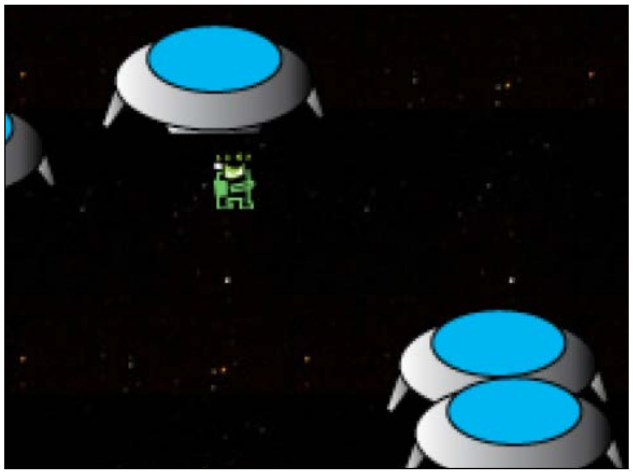
1. Once the `trigger` and `levelchange` entities are present in the `entities` folder, add both to the main script. Also add `level 2` to the script once you have created or copied it.

   ```
   'game.levels.level2',
   'game.entities.trigger',
   'game.entities.levelchange'
   ```

2. Put both the `trigger` and the `levelchange` entities in `level 1` using the Weltmeister.
3. Add a `name` property with the value `tolevel2` and a `level` property with the value `level2` to the `levelchange` entity using the Weltmeister.
4. Add a property called `target.1` with a value of `tolevel2` to the `trigger` entity using the Weltmeister.

5. Double check you have a player entity present in your second level and this level bears the name `level2`.
6. Save all changes you made and reload the game for a test run. Certainly try collecting a health crate before using the level transition. Once you arrive at `level2`, your health increase should persist.

If you copied `level2` from the downloadable files, notice how the stars move slower than the spaceships, which in turn move slower than some other spaceships. This is because of the distances of these three layers. If you open the Weltmeister, you can see that the star layer has a distance value of **5**, the closest star ships have a value of **2**, and the other ships have a value of **3**. Working with distances can give really nice effects for parallax games; use them wisely.



Adding a level transitioning can be done in a relatively easier manner, if it goes one way only. Let's recap how we did this:

- Copy the `trigger` and `levelchange` entities.
- Build or copy a second level called `level2`. Make sure to add a player entity to the level.
- Include the new level and both the `trigger` and `levelchange` entities in the main script.
- Add a `trigger` and `levelchange` entity to `level 1`, connect them and make sure the `levelchange` entity points to `level2`.
- Try using the `distance` property of a layer while designing your levels. This can give you beautiful results in a side scroller game.

The final battle

Every good game ends with a challenging final battle where good defeats evil, or vice versa, your call.

In order to have a challenging fight, let's create a separate `boss` entity, which is more powerful than our other frogs.

1. Open a new file and save it as `boss.js`.
2. The boss will be an extension to our normal enemy, so let's first define the characteristics on which he will differ from the red frogs.

```
ig.module(
  'game.entities.boss'
)
.requires(
  'game.entities.enemy'
)
.defines(function(){
  EntityBoss = EntityEnemy.extend({
  name: 'boss',
  size: {x: 32, y:48},
  health: 200,
  animSheet: new ig.AnimationSheet( 'media/Boss.png', 32,
48 )
});
});
```

3. His name is different; but more importantly, he will have more health and is much bigger than the other frogs.
4. Add the boss to your main script using the following line of code:

```
'game.entities.boss'
```

5. Save all changes and put the boss in one of your levels.

We now have a bigger enemy indeed, with more health, who basically does the same thing as the smaller ones. This doesn't really make for an interesting boss fight, so let's give him the ability to fire bullets like the player. We need a separate bullet entity, since our basic projectile can only damage type B entities and our player is of type A; also we might want it to look a bit different:

1. Open a new file and save it as `bossbullet.js`.
2. This bullet will be a direct extension of the normal bullet except for the type check and the way it looks. Write the following code to create the new bullet entity:

   ```
   ig.module(
     'game.entities.bossbullet'
   )
   .requires(
     'game.entities.projectile'
   )
   .defines(function(){
     EntityBossbullet = EntityProjectile.extend({
     name: 'bossbullet',
     checkAgainst: ig.Entity.TYPE.A,
     animSheet: new ig.AnimationSheet( 'media/bossbullet.png',
   8, 4 )
     });
   });
   ```

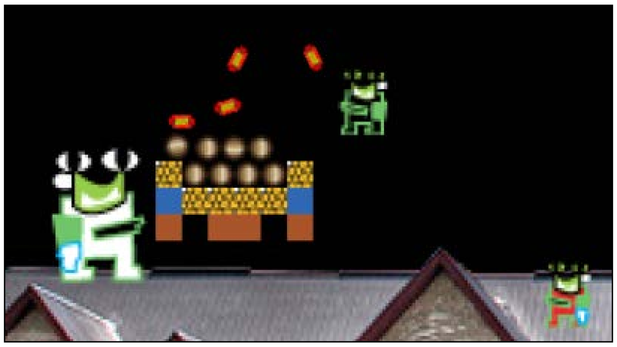
3. We need one last modification, shown in the following code, to make the boss fire his own bullets:

   ```
   update: function(){
     var players = ig.game.getEntitiesByType('EntityPlayer');
     var player = players[0];
     // both distance on x axis and y axis are calculated
     var distanceX = this.pos.x - player.pos.x;
     var sign = Math.abs(distanceX)/distanceX;
     var distanceY = this.pos.y - player.pos.y;
     //try to move without flying, fly if necessary
     if (Math.abs(distanceX) < 1000 &&Math.abs(distanceX)>
   100){
       var fY = distanceY> 0 ? -350: 0;
       this.body.ApplyForce( new b2.Vec2(sign * -
   50,fY),this.body.GetPosition() );
       if(distanceX>0){this.flip = true;}
       else {this.flip = false;}
       if (Math.random() > 0.9){
   ```

```
            var x = this.pos.x + (this.flip ? -6 : 6 );
            var y = this.pos.y + 6;
            ig.game.spawnEntity( EntityBossbullet, x, y,
{flip:this.flip} );
          }
          if(distanceY>= 0){this.currentAnim = this.anims.fly;}
          else{this.currentAnim = this.anims.idle;}
        }
        else if (Math.abs(distanceX) <= 100){
          if(Math.random() > 0.9){
            var x = this.pos.x + (this.flip ? -6 : 6 );
            var y = this.pos.y + 6;
            ig.game.spawnEntity( EntityBossbullet, x, y,
{flip:this.flip} );
          }
        }
        this.body.SetXForm(this.body.GetPosition(), 0);
        if (distanceX> 0){this.currentAnim.flip.x = true;}
        else{this.currentAnim.flip.x = false;}
        this.cooldowncounter ++;
        this.parent();
      }
```

4. The boss entity's `update()` function differs from the others in three main ways:
    ◦ Since he is a bigger creature, he uses more force to move around.
    ◦ We prefer him to deal damage with his bullets so he doesn't try to come into melee range. When he is within a distance of 1000 pixels on the x axis, he will approach. Once at a distance of 100 pixels, he doesn't try to come any closer.
    ◦ Last but not least, in every frame there is a 1 out of 10 chance of him firing a bullet. This should on an average result in 6 bullets every second, which is quite a barrage. If you are very unlucky, he can fire up to 60 bullets your way, in a single second.

A pretty nice effect from the Box2D collision is that as a player, your own bullets can deflect those of the boss. However, this will not always be the case. Collision detection in Box2D is not perfect yet and sometimes two entities can go straight through each other. This is also why you should make sure your outer boundary collision wall is pretty thick. Otherwise, you will have entities flying outside your level, possibly resulting in game crashes.



Killing the boss character should end the game and give the player a nice Victory screen. Dying should end in a Game Over screen instead of a Game Crash screen. These and many other things will be addressed in *Chapter 5*, *Adding Some Advanced Features to Your Game*, where we take a deeper look at some more advanced features with which you can enhance your game.

When the game is almost at an end, the player expects a climax. This can be given to him in the form of an epic battle against a worthy foe. That is exactly what we did earlier in this chapter. The boss character is the player's ultimate enemy and his ticket to victory:

- Open a new file and save it as `boss.js`.
- Create the boss character's basics as an extension of the enemy entity.
- Introduce the boss' bullet, that is, the projectile used by the boss to kill the player. This is an extension of the projectile used by the player himself.
- Upgrade the boss so he can make use of his lethal new bullet.
- Add a boss to your game and check out if you are able to defeat him.

Summary

In this chapter we learned about the side scroller game and took a look at some famous examples. We built our own side scroller using Box2D, that is, a physics engine integrated with ImpactJS.

First, we built a level with the Weltmeister, so we could populate them with our newly created enemy and playable characters. Lifeless crates were added to fully display the physics of Box2D. In order to arm the player against the violent enemies we introduced pickup items and two interesting weapons, that is, the bullet and the bomb.

Our enemy was given life once we added minor AI. As a final challenge for the player, the formidable boss was brought to the scene. This enemy is stronger than regular enemies and capable of firing bullets like the player. For defeating every enemy, the player is awarded with extra points to his total score.

In the next chapter we will explore some new concepts like working with data and go deeper into some of the features we have already touched on, such as debugging AI.

5
Adding Some Advanced Features to Your Game

In the previous chapters we saw how to set up a work environment, took a look at the Impact engine, and even built two types of games. Now it is time to have a look at a few interesting extras.

To test the elements covered in this chapter, it is best to either download the code material in the `chapter 5` folder or build directly on the game we designed in *Chapter 3*, *Let's Build a Role Playing Game*. Since we will not be working with the Box2D extension in this chapter, some things will be incompatible with the side scroller game in *Chapter 4*, *Let's Build a Side Scroller Game*. In this chapter we will cover:

- Making a Start and Victory screen
- Extra debug possibilities and introducing a customized ImpactJS debug panel
- Saving data with cookies and the lawnchair application, and turning an Excel file into useful game data
- A few extra game functionalities for the role playing game (RPG) in *Chapter 3*, *Let's Build a Role Playing Game*
- Making your character move by use of a mouse
- Intelligent spawn locations
- Adding basic conversation
- Showing a player's health bar
- Extending the Artificial Intelligence (AI) with a hive mind
- Implementing Playtomic for game analytics

The Start and Game-over screen

When the player starts your game, the first thing you might want him to see is a splash screen. This screen usually has the name of the game and other interesting info; often it contains some information on the game's story or controls. At the end of the game you could have a Victory screen that tells the player how well he did by placing his achieved score in a leaderboard.

With regards to code, this can be done by introducing new game instances next to the actual game. Every single screen: Start, Game-over, and Victory are direct extensions of the ImpactJS game class. Let's kick off by creating a Start screen.

The game's Start screen

In order to make a nice opening screen, we need a background image and our trusted `main.js` script:

1. Open the `main.js` script and insert the following code:

```
OpenScreen = ig.Game.extend({
  StartImage : new ig.Image('media/StartScreen.png'),
  init:function(){
  if(ig.ua.mobile){
    ig.system.setGame(MyGame);
  }
    ig.input.bind(ig.KEY.SPACE,'LoadGame');
  },
  init:function(){
    if(ig.ua.mobile){ig.input.bindTouch( '#canvas',
'LoadGame' );}
    else {ig.input.bind(ig.KEY.SPACE,'LoadGame');}
  },
```

2. The opening screen is an extension of the `ig.Game` function, just like our game is. As a matter of fact, when we are done here, we will have four game instances: one real game called `MyGame`, and three other games, which will merely act as a Start, Victory, or Game-over screen. This might be a little counter intuitive, since you would expect the screens to be part of that same game. In reality this is most certainly true. However, in code it is more convenient to turn these screens into separate game class extensions.
3. In this part of the `OpenScreen` code, we first define the image we will be showing: `StartScreen.png`.

4. Finally we bind the Space bar to an action state called `LoadGame` shown as follows:

   ```
   update:function(){
     if(ig.input.pressed('LoadGame')){
       ig.system.setGame(MyGame);
     }
   },
   ```

5. Now we can load the game by pressing the Space bar, but we still need to actually show something on the screen.
6. We can visualize things by manipulating the `draw()` function, as shown in the following code snippet, of any ImpactJS class:

   ```
     draw: function(){
       this.parent();
       this.StartImage.draw(0,0);
       var canvas = document.getElementById('canvas');
       if(canvas.getContext){
         var context = canvas.getContext('2d');
         context.fillStyle = "rgb(150,29,28)";
         context.fillRect (10,10,100,30);
       }
       var font = new ig.Font('media/font.png');
       font.draw('player:' + GameInfo.name,10,10);
     }
   }),
   ```

7. The `draw()` function will draw the background image we specified when initializing our `OpenScreen` function. Having done so, it will also add a small red rectangle in which we will print the name of the player, if we have it. We will go into getting this name and storing it for later use when we take a look at game data later in this chapter. For now, the `GameInfo.name` variable is undefined and will show up exactly like starting a new game.
8. To make sure our brand new opening screen is actually used, we need to replace the `MyGame` game class instance by the `OpenScreen` function in our `ig.main` function call, as shown in the following code line:

   ```
   ig.main( '#canvas', OpenScreen, 60, 320, 240, 2 );
   ```

We now have an opening screen! Adding a Game-over screen and a Victory screen are very similar procedures. Before making these other screens, let's quickly recap on what we have just done:

- We made sure a background image is available in the `media` folder
- We added the `OpenScreen` function as a new game instance
- We bound the Space bar so it can be used to load the actual game
- We set up the `Draw()` function so it can show the background and even the player's name later on
- We initialized our canvas in the `OpenScreen` function window instead of the `MyGame` game class instance

The Victory and Game-over screens

The Victory screen is a relatively simple extension of the game entity. The procedure is pretty much the same for every type of screen we will want to show. To set up the Victory screen, follow these steps:

1. Open the `game.js` file and add our new `GameEnd` game class, as shown in the following code:

```
GameEnd = ig.Game.extend({
  EndImage : new ig.Image('media/Winner.png'),

  init:function(){
    if(ig.ua.mobile){ig.input.bindTouch( '#canvas',
'LoadGame' );}
    else {ig.input.bind(ig.KEY.SPACE,'LoadGame');}
  },
```

2. All we need to initialize here is the image we will be showing and a key for restarting our game.
3. Similar to the Start screen, we use the Space bar to load a new game. We continually check whether the Space bar was pressed by adding the following `if` statement to the `update` function:

```
  update:function(){
    if(ig.input.pressed('LoadGame')){
      ig.system.setGame(MyGame);
    }
  },
```

4. We need to draw the actual game-end image and place the text **HIT SPACE TO RESTART** using the following code. This way we make sure the player doesn't resort to refreshing the browser instead of using the Space bar.

```
  draw: function(){
    this.parent();
    var font = new ig.Font('media/font.png');
    this.StartImage.draw(0,0);

  if(ig.ua.mobile){
    font.draw('HIT THE SCREEN TO RESTART:',100,100);
  }
else font.draw('HIT SPACE TO RESTART:',100,100);
  }
}),
```

5. The Victory screen needs to be shown when the player is at the end of the game. In our case, this would be when the boss entity is defeated. Open the `boss.js` file and change the `kill()` method as shown in the following code so that he will load the Victory screen when he dies:

```
kill: function(){
  ig.game.gameWon();
}
```

6. In the `kill()` method, we call the `gameWon()` function, which is a method of our current game and which is not yet defined.
7. Open the `game.js` file and add the `gameWon()` method as a new method of the `MyGame` file, as shown in the following code.

```
gameWon: function(){
  ig.system.setGame(GameEnd);
}
```

8. At the moment, it may seem rather pointless to introduce an extra intermediary function to call the Victory screen. However, this will start to make sense once we get into handling game data. Eventually, this function will not only call the Victory screen, but will also save the player's score. Using an intermediary function is a cleaner way of programming than adding the `ig.system.setGame()` function directly into the player entity.

The Game-over screen can be the exact equivalent of the Victory screen, except that another image is used and it is triggered by the death of the player, not the boss.

9. Add the `gameOver` function to the `game.js` file as shown in the following code snippet:

```
gameOver = ig.Game.extend({
  gameOverImage : new ig.Image('media/GameOver.png'),
  init: function(){
    ig.input.bind(ig.KEY.SPACE,'LoadGame');
  },
  update:function(){
    if(ig.input.pressed('LoadGame')){
      ig.system.setGame(MyGame);
    }
  },
  draw: function(){
    this.parent();
    var font = new ig.Font('media/font.png');
    this.gameOverImage.draw(0,0);
    font.draw('HIT SPACE TO RESTART',150,50);
  }
}),
```

10. Make sure the `gameOver` function is triggered by the death of our player by adjusting his `kill()` method using the following code:

```
kill: function(){
    ig.game.gameOver();
}
```

11. Again we called an intermediary function to handle the actual screen loading. This function will need to be added as a method to the `MyGame` game class instance.
12. In the `game.js` script, add the `gameOver()` method to the `MyGame` game class instance as shown in the following code:

```
gameOver: function(){
  ig.system.setGame(gameOver);
},
```

These are very basic Start and Game-over screens, which show that it can be done by using the `ig.game` class as a starting point. A good idea for both the Victory and Game-over screen would be to show a leaderboard or any other interesting info gathered during the game.

When the game gets more complicated by adding advanced features, debugging becomes increasingly important to cope with these added levels of complexities. We will now have a look at what advanced debugging options we have at our disposal. However, before we do that, let's quickly recap on the Victory and Game-over screens:

- We made two new game instances that function as Victory and Game-over screens
- The `update` function was adapted to listen to the Space bar, while the `draw` function was adapted to show both the background image and the **HIT SPACE TO RESTART** message
- The functions of both the boss and player entity were adapted to trigger the Victory and Game-over screens
- We made use of intermediary functions called `gameOver()` and `gameWon()` because we wanted to adapt these later on so that they trigger the lawnchair application to store a score

More advanced debugging options

In *Chapter 1*, *Firing Up Your First Impact Game*, we took a look at how to debug using the browser and what the ImpactJS debug panel has to offer. Here we will go a little further by making ourselves a new ImpactJS debug panel. This code is readily made available by Dominic on his ImpactJS website but many people overlook this functionality even though it is surprisingly useful.

In *Chapter 1*, *Firing Up Your First Impact Game*, we also talked about the logical error, an error that is very difficult to find because it does not necessarily generate an error in the browser debug console. To cope with these kinds of errors, programmers often use a method known as unit testing. This basically involves defining in advance what your desired result is for every piece of code, translating these desired results into conditions, and testing whether the output adheres to these conditions. Let's take a look at a short example.

Short introduction to unit testing

One of the most basic components of our ImpactJS scripts is the function. Some of our functions return values, others alter properties directly. Let's assume we have a function called `dummyUnitTest()`, which takes a single argument: `functioninput`.

```
dummyUnitTest: function(inputnumber){
  var outputnumber= Math.pow(inputnumber,2);
  return null; // can cause an error in subsequent
functions,comment out to fix it
  return outputnumber;
}
```

An `inputnumber` variable can be any number, but our function transforms the `inputnumber` variable into an `outputnumber` variable, which it then returns. The `inputnumber` variable raised to the power of two should always return a positive number. So we can say at least two things about what we expect from this function: the output cannot be null and it cannot be negative.

We can unit test this function by adding `assert` functions dedicated to checking certain conditions. An `assert` function checks a condition and when it is false, it will write a message to the console log. The console element itself has this function but so does ImpactJS when the debug module is activated. The `ig.assert()` function is the ImpactJS equivalent of the `Console.assert()` function. Remember that activating ImpactJS debugging can be done by including `'impact.debug.debug'` in the `main.js` file. Using the `ig.assert` function is preferable over the `console.assert()` function. This is because getting rid of the `ig` class messages when preparing to launch the game is done simply by turning off the ImpactJS debug module. Methods of the console class, such as the `console.assert()` call would need to be turned off separately. In general, the `assert()` function looks like this:

```
ig.assert(this.dummyUnitTest('expected')==='expected','you introduced
a logical error you should retrieve the same value as the input');
```

For our specific example we could perform several tests as shown in the following code:

```
ig.assert(typeof argument1 === 'number','the input is not a number');
ig.assert(typeof argument2 === 'number','the output is not a number');
ig.assert(typeof argument2 >= 0,'the output is negative');
ig.assert(typeof argument2 != null,'the output is null);
```

We could go on, and this method is not without the pitfall of overkill. But in general, when you plan on building a very complex game, unit testing can help out tremendously by reducing the time you will spend looking for the source of logical errors. For example, if in this case our output was a negative number, the function itself will not fail; maybe most of the code that relies on this function will not either, but somewhere down the chain, something will. While introducing all these dependencies, one function built on top of another and so on, unit testing is certainly justified.

Next to the `ig.assert()` and `ig.log()` function is another interesting function. It is the ImpactJS equivalent of the `console.log()` function and will write to the log at all times, without checking certain conditions. This can be useful for keeping an eye on an enemy's health without having to look for it in the **Document Object Model** (**DOM**).

Let's quickly recap what unit testing is about before moving on to our own ImpactJS debug panel:

- Unit testing is about foreseeing what you expect your code components to do, and returning and checking the output validity.
- We use the `ig.assert()` or `console.assert()` function to check certain conditions and print a message to the log if they are violated.

Adding your own debug panel to the ImpactJS debugger

As stated earlier, activating the debug panel is done by simply including the `'impact.debug'` statement in the `main.js` file. When starting a new game, the panel is minimized at the bottom of the screen and can be made entirely visible by simply clicking on it.

Let's get round to building our own panel, which will enable us to activate and deactivate entities while playing the game. This way we can venture unharmed, past our most ferocious enemies by freezing them in position. Let's get to it:

1. Open a new file and save it as `MyDebugPanel.js`.
2. Insert the following code in the file:

```
ig.module(
  'plugins.debug.MyDebugPanel'
)
.requires(
  'impact.debug.menu',
  'impact.entity',
  'impact.game'
)
.defines(function(){
ig.Game.inject({
  loadLevel: function( data ) {
    this.parent(data);
    ig.debug.panels.fancypanel.load(this);
  }
})
})
```

3. Before we actually define our panel, we will inject code in two ImpactJS core classes: `Game` and `Entity`. Injecting code is like extending, except we don't make a new class. The original code is replaced by an extended version of itself. In the previous code we tell the core `loadlevel()` function to load our panel as well, which will be called a **Fancy panel**.

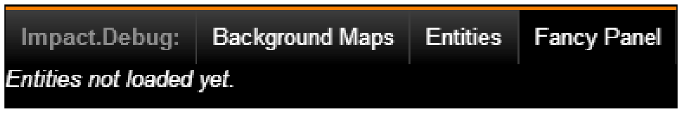
4. We then upgrade our core entity code by adding a new property: `_shouldUpdate`, as shown in the following code:

   ```
   ig.Entity.inject({
     _shouldUpdate: true,
     update: function() {
       if( this._shouldUpdate ) {
       this.parent();
       }
     }
   });
   ```

5. When true, the entity's `update` method will be called, which is also the default method. However, when false, the `update()` function is bypassed and no real action will be performed by the entity.
6. Now let's have a look at the panel itself. We can see the following code contained in the panel:

   ```
   MyFancyDebugPanel = ig.DebugPanel.extend({
     init: function( name, label ) {
       this.parent( name, label );
       this.container.innerHTML = '<em>Entities not loaded
   yet.</em>';
     },
   }
   ```

7. Our fancy panel is initialized as an extension of the panel of ImpactJS, called the `DebugPanel`. Calling the `this.parent` function will make sure a DIV container is supplied to the panel so that it can be shown in HTML5. The container would not hold anything if there were no entities in the game, so then a message is placed instead. For example, this will be the case for our start and end screens. Since the `this.container.innerHTML` function will hold the panel's content, opening the panel while in the start screen should result in the message **Entities not loaded yet**.



Entities not loaded yet.

To display the previous message we should add the following code within the `this.container.innerHTML` function:

```
load: function( game ) {
  this.container.innerHTML = '';
    for( var i = 0; i < game.entities.length; i++ ) {
      var ent = game.entities[i];
      if( ent.name ) {
        var opt = new ig.DebugOption( 'Entity ' + ent.name,
ent,'_shouldUpdate' );
        this.addOption( opt );
        this.container.appendChild(document.createTextNode
('health: '+ ent.name + ' :' +ent.health));
      }
    }
},
```

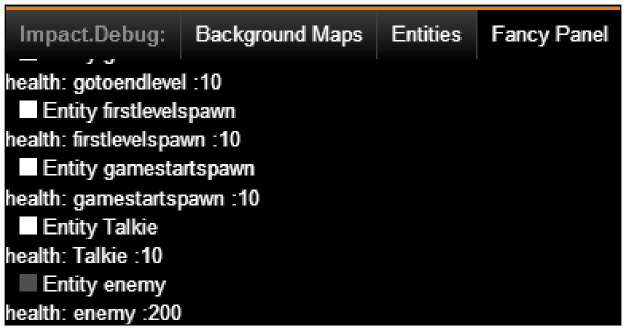
8. On loading the level, our panel is filled with all the entities in the game and the option to turn their `update()` functions off. Also, their health is shown. The `addOption()` function is what makes it possible to switch the `_shouldUpdate` property from true to false and back, whenever needed. It takes two arguments: a label and the variable that needs to alternate between true and false.
9. These last functions are not used for our particular panel but are useful nonetheless. The following code explains the previous functions:

```
ready: function() {
  // This function is automatically called when a new game
is created.
  // ig.game is valid here!
},
beforeRun: function() {
  // This function is automatically called BEFORE each
frame is processed.
},
afterRun: function() {
  // This function is automatically called AFTER each frame
is processed.
}
});
```

10. The main difference between the `load()`, `ready()`, `beforeRun()`, and `afterRun()` function is the moment they are called within the game. Depending on your needs, you will use one, the other, or a combination. We used the `load()` method, which is called when a level is loaded. But for other panels you might want to use the other methods.
11. As a last step, we actually add the customized panel to our set of standard panels as shown in the following code:

    ```
    ig.debug.addPanel({
      type: MyFancyDebugPanel,
      name: 'fancypanel',
      label: 'Fancy Panel'
    });
    ```

12. Reload your game and take a look at your new panel. Try freezing your enemies! You will notice that the enemies will still face the player but not move towards him. This is because we disabled their `update()` method but not their `draw()` methods.



```
health: gotoendlevel :10
 ■ Entity firstlevelspawn
health: firstlevelspawn :10
 ■ Entity gamestartspawn
health: gamestartspawn :10
 ■ Entity Talkie
health: Talkie :10
 ■ Entity enemy
health: enemy :200
```

We will now move on to using game data, but let's first have a look at what we just covered:

- ImpactJS has a very interesting debugger for which you can design your own panels.
- Activating the ImpactJS debugger is done by including the `'impact.debug.debug'` command in your main script.
- We made ourselves a panel by extending the ImpactJS `DebugPanel` class. Our own panel needs to enable us to freeze any entity into position, so we can explore our levels unimpeded.

- Making use of a technique called injection; we changed our core entity class so that the debug panel gets to control the `update` function of every entity.
- Finally we added our debug panel to the standard set so that it becomes available to us at all times.

Handling game data

Working with data can be essential for game building. Simple games do not need explicit data management. However, when we start looking at games that feature conversations or keep high-scores, understanding data handling becomes an important topic. We will talk about two things:

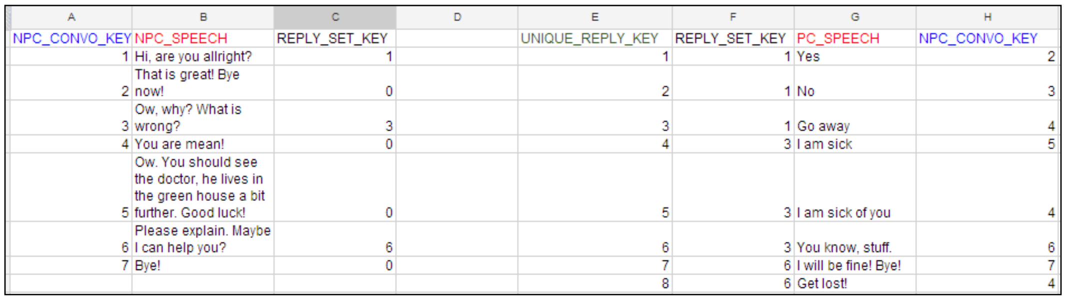
- Bringing data into your game
- Storing data that is generated in a game

For the latter we will have a look at two different ways of tackling the problem: cookies and the lawnchair application.

Let's first have a look at what we need to do if we want to introduce data for a conversation between an NPC and a player.

Adding data to your game

As mentioned before, RPGs are often packed with conversations between the player and several non-playable characters (NPCs). In these conversations the player is given several options when it is his time to answer. The code mechanics for this can get pretty complicated and we will get to that later in this chapter, but first we need the actual sentences themselves. We can prepare these in an application such as Excel.



A	B	C	D	E	F	G	H
NPC_CONVO_KEY	NPC_SPEECH	REPLY_SET_KEY		UNIQUE_REPLY_KEY	REPLY_SET_KEY	PC_SPEECH	NPC_CONVO_KEY
1	Hi, are you allright?	1		1	1	Yes	2
2	That is great! Bye now!	0		2	1	No	3
3	Ow, why? What is wrong?	3		3	1	Go away	4
4	You are mean!	0		4	3	I am sick	5
5	Ow. You should see the doctor, he lives in the green house a bit further. Good luck!	0		5	3	I am sick of you	4
6	Please explain. Maybe I can help you?	6		6	3	You know, stuff.	6
7	Bye!	0		7	6	I will be fine! Bye!	7
				8	6	Get lost!	4

Setting up RPG conversations is an art; there are many ways to do it, each with its pros and cons. Creating a decent conversation setup and flow, even database-wise, is a discussion that reaches beyond the scope of this book. Here we will try to keep it simple and work with two tables: one for all the things an NPC can say and one for what the player can answer. The flow of a conversation for our game will be as follows:

1. The NPC says something. Everything an NPC can say has a unique key called **NPC_CONVO_KEY**.
2. The player is presented with a set of possible answers. Every set gets a key called **REPLY_SET_KEY**. In addition to that, although it will not be used by us, every answer has its own unique key, which we call **UNIQUE_REPLY_KEY**. It is essentially a good practice to have primary keys available, even if you don't use them just yet.
3. The player selects one of the answers. An answer has a foreign key, which leads back to the NPC. We named this foreign key: **NPC_CONVO_KEY**.
4. Using **NPC_CONVO_KEY**, the NPC knows what to say next, and we have completed the loop. This will go on until the conversation is abruptly aborted or naturally comes to an end.

The actual sentences are kept in the variables **PC_SPEECH** and **NPC_SPEECH**.

We can easily prepare our data in an Excel document but we still need to import it into our game. We will do this by using a converter such as the one on the following website: `http://shancarter.com/data_converter/`.

Simply copy and paste the data from Excel to the converter and select **JSON-Column Arrays** to get your data in the JSON-format documents.

Once we have it in this format, all we need to do is copy and paste the data to separate modules. The following code is what our excel data looks like once it is converted to JSON:

```
ig.module('plugins.conversation.npc_con')
.defines(function(){
npc_con=/*JSON[*/{
  "NPC_CONVO_KEY":[1,2,3,4,5,6,7],
  "NPC_SPEECH":["Hi, are you allright?","That is great! Bye now!",
"Ow, why? What is wrong?",
"You are mean!",
"Ow. You should see the doctor, he lives in the green house a bit
further. Good luck!",
"Please explain. Maybe I can help you?","Bye!"],
  "REPLY_SET_KEY":[1,0,3,0,0,6,0]
}
});
```

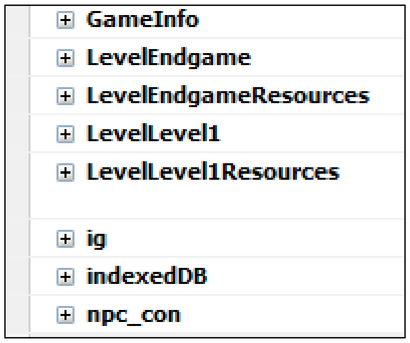
We store the data in JSON format just like the Weltmeister does with the level files. The following code is the player's speech data turned into JSON:

```
ig.module( 'plugins.conversation.pc_con' )
.defines(function(){
pc_con=/*JSON[*/{
  "UNIQUE_REPLY_KEY":[1,2,3,4,5,6,7,8],
  "REPLY_SET_KEY":[1,1,1,3,3,3,6,6],
  "PC_SPEECH":["Yes","No","Go away",
"I am sick","I am sick of you",
"You know, stuff.","I will be fine! Bye!",
"Get lost! "],
  "NPC_CONVO_KEY":[2,3,4,5,4,6,7,4]
}
});
```

All that remains now is putting the data inside our game directory and including both files in the `main.js` file:

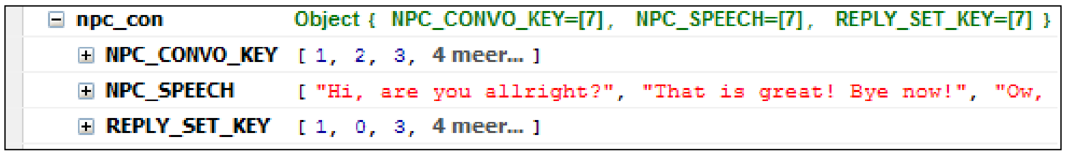
```
'plugins.conversation.npc_con',
'plugins.conversation.pc_con',
```

If you were to reload your game, you should be able to explore your newly introduced data in the DOM with the Firebug application as shown in the following screenshot:



```
⊞ GameInfo
⊞ LevelEndgame
⊞ LevelEndgameResources
⊞ LevelLevel1
⊞ LevelLevel1Resources

⊞ ig
⊞ indexedDB
⊞ npc_con
```



```
⊟ npc_con           Object { NPC_CONVO_KEY=[7], NPC_SPEECH=[7], REPLY_SET_KEY=[7] }
   ⊞ NPC_CONVO_KEY  [ 1, 2, 3, 4 meer... ]
   ⊞ NPC_SPEECH     [ "Hi, are you allright?", "That is great! Bye now!", "Ow,
   ⊞ REPLY_SET_KEY  [ 1, 0, 3, 4 meer... ]
```

Now that we have taken a look at how to introduce data, let's take a look at two ways to store data on the player's computer, starting with cookies. But let's first summarize what we have done here:

- Setting up conversations is an art form, which will not be explored in depth in this chapter
- We set up a simple conversation in Excel or an equivalent application
- This Excel sheet is converted to a JSON-format document. You can do this by using online converters such as the one on `http://shancarter.com/data_converter/`
- We turn the new JSON encoded data into ImpactJS modules
- Finally we include the two newly created data modules in our main script

Using cookies to store data on the player's computer

Cookies are not much more than a piece of string data stored in your browser and are used by many websites to track visitors. If you are a user of Google Analytics, you probably know that Google provides a script that places several different cookies per visitor. Google Analytics is not the only program that works this way. After a good day of surfing the Internet, your browser is full of cookies; some of them will remain there for several months until finally deleting themselves.

Storing information such as the player's name and high score in the user's browser makes sense; no storage is needed from your side, thus, no need for PHP or SQL coding. The downside is the loss of data if the player decides it's time to clean up his browser. Also, there is no real one-to-one relationship with the player while using cookies. One person can have several devices and even several browsers per device. Cookies are thus recommended for games you always replay from the start. It is certainly not good for games that require a massive time investment from the player; for instance, MMORPGs (Massively Multiplayer Online Role Playing Games) tend to do. For these more advanced games, working with accounts and server-side databases is the way to go.

Let's build ourselves a cookie plugin, using the following steps, which will be able to store the player's name so that we can retrieve it when restarting the game:

1. Open a new file and save it as `cookie.js`. Insert basic class extension code as follows:

   ```
   ig.module('plugins.data.cookie').
     defines(function(){
       ig.cookie = ig.Class.extend({
       userName : null,
       init: function(){
         this.checkCookie();
     },
   ```

2. We start off by defining our cookie plugin as an ImpactJS class extension. We know it will have to store a username later on, so let's initialize it with the value `null`. The first thing our new DOM object will do, when created, is call the `checkCookie()` function. The `checkCookie()` function will check if there is already a cookie with the same username stored inside. There are of course two possibilities here: either it exists or it doesn't. If it doesn't, the name needs to be prompted and stored. If the username was stored before, it can be retrieved.
3. Putting a cookie in place is done with the `setCookie()` function as shown in the following code:

   ```
   setCookie: function(c_name,value,exdays){
     var exdate=new Date();
     exdate.setDate(exdate.getDate() + exdays);
     var c_value=escape(value) + ((exdays==null) ? "" : ";
   expires="+exdate.toUTCString());
   document.cookie=c_name + "=" + c_value;
   },
   ```

4. This function takes in three arguments:
   - `c_name`: The name of the variable it needs to store, which is the username
   - `value`: The value of the username
   - `exdays`: The number of days the cookie is allowed to exist until it should delete itself from the browser

5. The `setcookie()` function serves to check the input data's validity. The value is transformed, so it's more difficult for an amateur hacker to insert damaging code instead of a name. The data is then stored in the `document.cookie` variable, a part of the DOM that stores all the cookies and is not lost while closing the page. Going into the intricate working of the `document.cookie` variable will lead us too far but it behaves very peculiarly. Assigning a value to the `document.cookie` variable, as shown in the previous code snippet, will not replace what was already in there with the newly assigned value. Instead, it will be added to the rest of the stack.
6. If there is a `setCookie()` function, there is of course a `getCookie()` function as shown in the following code snippet:

```
getCookie: function(c_name){
  var i,x,y,ARRcookies=document.cookie.split(";");
  for (i=0;i<ARRcookies.length;i++){
    x=ARRcookies[i].substr(0,ARRcookies[i].indexOf("="));
    y=ARRcookies[i].substr(ARRcookies[i].indexOf("=")+1);
    x=x.replace(/^\s+|\s+$/g,"");
    if (x==c_name){
      return unescape(y);
    }
  }
},
```

7. The previous code will decode the transformed cookie and return it. Its sole input argument is the name of the variable you are looking for.
8. In programming, especially in Java, it is very common to use a combination of the `set` and `get` functions to change properties. So using this programming logic, a `health` property for instance should always have a `setHealth()` and `getHealth()` function. Changing parameters directly has advantages and disadvantages. The main advantage of directly changing properties is pragmatism; things remain easy and intuitive to understand. A big downside is the struggle to maintain code validity. If everything can just change any property of any entity from anywhere, you can end up with serious issues if you lose sight of things.
9. The `checkCookie()` function checks if the username is present in the browser by use of the `getCookie()` function:

```
checkCookie :function(){
  var username=this.getCookie("username");
  if (username!=null && username!=""){
  this.setUserName(username);
  }
  else {
    username=prompt("Please enter your name:","");
    if (username!=null && username!=""){
      this.setCookie("username",username,365);
    }
  }
},
```

10. If a cookie is present, a `setUserName()` function is called using the fetched username as an input parameter. If no cookie is present, the player is prompted to insert his or her name and this is subsequently stored with the `setCookie()` function.
11. The `getUserName()` and `setUserName()` functions are kept relatively basic for this example, as shown in the following code:

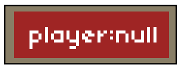
```
getUserName: function(){
  return this.userName;
},
setUserName: function(userName){
  if(userName.length > 10){alert("username is too long");}
  else { this.userName = userName; }
}
```

12. The `setUsername()` and `getUsername()` functions could be left out by directly getting or setting the `this.username` command with the `checkCookie()` and `setCookie()` functions. However, as said earlier, it is a good programming practice to use the `set` and `get` statements wherever a property needs to be changed. As seen in the `setUserName()` function, some extra checks can be built into these functions. While the `getCookie()` and `setCookie()` functions make sure the data is stored in a harmless way and fetched appropriately, the `setUserName()` and `getUserName()` functions can be used to check other constraints, such as name length.
13. Now that we have finished our cookie extension, we can actually make use of it. Open the `main.js` file and add the following two lines to the `GameInfo` class:

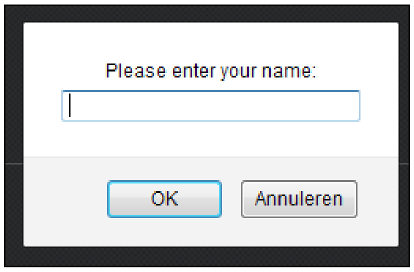
```
this.cookie = new ig.cookie();//ask username or retrieve if
not set
this.userName = this.cookie.getUserName();//store the
username
```

14. The `GameInfo` class is perfect for this; everything we want to remain available outside of a game instance needs to be gathered in the `GameInfo` class. Separating the data component from the game logic as much as possible is a way of keeping things clean and understandable when a game grows more complex.

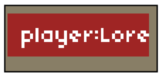
15. Our first code line will create an `ig.cookie` array and immediately check if a username is present. If it is not present, a prompt will appear and the name will be stored after the player has filled out the prompt alert.
16. The second line simply hands over the username to the `GameInfo` object that we first encountered in *Chapter 3, Let's Build a Role Playing Game*. As you might remember, we used the `GameInfo.name` variable in the beginning of this chapter but it was undefined. Now it will be set to `null` until the player gives his name and is, henceforth, used for every game he plays.



Initially the player's name will be unknown, and **null** would be shown on the screen as shown in the previous screenshot.



However, the player is prompted to fill out his or her name in the window as shown in the previous screenshot.



Henceforth, the real name is shown on the screen as shown in the previous screenshot.

While you should be able to get around using cookies, there is another way to store data, which is probably more versatile and easy to use: lawnchair. The lawnchair application makes use of the HTML5 local storage, also known as the DOM storage. Before moving on to the lawnchair application, we will take a quick peek into how to use HTML5 local storage even without using the lawnchair application:

- Cookies are a way of storing data in the player's browser. Many websites make use of them including the web analytics platform, Google Analytics. Cookies are useful for short games that are meant to be played of and over again, not for complicated games that require many things to be stored over a long period of time.

- We can implement the use of cookies by creating a `cookies` plugin. This plugin, once activated, will check whether a cookie is already active and place one if none is found.
- In this example, we use cookies to store and retrieve the player's name, which we first ask him to fill out if no cookie is in place.
- Emphasis is put on the use of the `set()` and `get()` functions. These functions are a standard practice in Java and are a useful technique to keep sight of things and check the validity of any property in the code that grows even more complex.

Local storage

Local storage, also known as DOM storage, is an HTML5 feature that allows you to save information on the user's computer. It is superior to cookies in almost every way but the older browsers do not support it. Using local storage is rather straightforward as we can see in the following code snippet:

```
ig.module('plugins.data.local').
defines(function(){
  ig.local = ig.Class.extend({
    setData: function(key, data){
      localStorage.setItem(key, data);
    },
    getData: function(key){
      return localStorage.getItem(key);
    }
  });
})
```

This plugin is not strictly necessary in order to make use of local storage. It is merely an extension with the `get` and `set` technique in order to check data validity. You can use the plugin by including the `'plugins.data.local'` command in your `main.js` script and calling the `setData()` and `getData()` functions.

```
Ls = new ig.local(); //localstorage
  Ls.setData("name","Davy");
  alert(Ls.getData("name"));
```

Now we'll take a quick look at how to use local storage in general; let's look at what the lawnchair application has to offer.

Using lawnchair as a versatile way of storing data

The lawnchair application is a free and very professional solution for storing data on the client side. It is capable of storing data in several ways, and a plugin for ImpactJS is readily available. Let's look at how to use the lawnchair application to store data:

1. Download the lawnchair application from the following website: `http://brian.io/lawnchair/`, or you can download the ImpactJS adapted version on `https://github.com/jmo84/Lawnchair-plugin-for-ImpactJS`.
2. Place the files in your `plugin` folder. In this example, they are placed in separate subfolders called `data` and `Lawnchair`. However, you are free to use any structure you want, as long as you make sure to change the code accordingly.
3. Include the `impact-plugin` file in your `main.js` file as shown in the following code:

   ```
   'plugins.data.lawnchair.impact-plugin',
   ```

4. Add a store element to your `GameInfo` class by the use of the newly acquired `ig.Lawnchair()` method as shown in the following code line:

   ```
   this.store = new ig.Lawnchair
   ({adaptor:'dom',table:'allscores'}
   ,function() { ig.log('teststore is ready'); }),
   ```

   The `ig.Lawnchair()` method takes two input parameters:

   ◦ The first parameter is the most important one and is actually an array. In this array you will need to specify two things: which method you want to use to store everything, and the name of the data storage you want to create. The first variable is called the `adaptor` because the lawnchair application uses the adaptor pattern technique to decide what needs to happen next. The lawnchair application is programmed very efficiently and this immediately becomes apparent by this use of patterns. An adaptor pattern is essentially a piece of code that will link your own code to that of the lawnchair application's storing system. Without this pattern, it would be really difficult to communicate with the actual lawnchair application's source code, which saves your data. Here we choose to save it as a permanent DOM storage, but other options such as Webkit-SQLite are available.

Webkit-SQLite differs from permanent DOM storage such that it acts more like a regular database, but on the client's local storage. For instance, like other databases, you can query Webkit-SQLite storage with SQL.

 - The second input parameter is an optional one. Here you can put functions that need to be executed when the `store` variable is prepared. This is the perfect place to put log messages.

5. Now that we have our store element ready to go, it is just a matter of storing whatever data you want by calling the `store.save()` method. Let's say we want to store our player's score. For this we can add a method to our `GameInfo` class that does just the same.

   ```
   this.saveScore = function(){
     this.store.save({score:this.score});
   }
   ```

6. The `saveScore()` function can be added to both the `gameOver()` and `gameWon()` methods we created while building our victory and Game-over screens as follows:

   ```
   gameOver: function(){
     GameInfo.saveScore();
     ig.system.setGame(gameOver);
   },
   gameWon: function(){
     GameInfo.saveScore();
     ig.system.setGame(GameEnd);
   }
   ```

7. When the player dies or wins the game, his score is saved using the lawnchair permanent DOM method. Permanent DOM does not mean the DOM is saved permanently on the user's PC; it is merely another name for local storage.
8. The last important thing we need to be able to do is retrieve the data. To do this, we introduce three new functions to our `GameInfo` class:
   - The `setScore()` function will save the input parameter as a `GameInfo.score` class if it is an actual number, as shown in the following code:

     ```
     this.setScore = function(score){
       if(typeof score == 'number')
       this.score = score;
     };
     ```

   - The `getScore()` method will just return the `score value` stored in the `GameInfo.score` class as shown in the following code:

     ```
     this.getScore = function() {
       return this.score;
     };
     ```

> Neither `setScore()` nor `getScore()` seem too important, but as explained while looking at the concept of cookies, using the `set` and `get` statements are useful for doing checks on data validity.

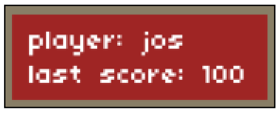
- The `GameInfo.getSavedScore()` method is the mirror opposite of the `GameInfo.saveScore()` method, as we can see in the following code:

```
this.getSavedScore = function(){
  this.store.get('score',
function(score){GameInfo.setScore(score.value) });
  return this.getScore();
};
```

9. The `getSavedScore()` method makes use of the `setScore()` function to set the `GameInfo.score` class to the number it has withdrawn from the storage. Then it returns this score using the `getScore()` method where some extra tests can be done on data validity.
10. You can now retrieve the last achieved score whenever you want!
11. We can adapt our opening screen so that it shows the last achieved score by adding the following code line to its `draw()` function.

```
font.draw('last score: ' + GameInfo.getSavedScore(), 10,20);
```

The last score of the player is displayed as shown in the following screenshot:



```
player: jos
last score: 100
```

Enough about data storage, let's quickly go over the differences between cookies, local storage, and the somewhat more versatile way of using local storage: lawnchair.

	Storage size	**Expiration date**	**Information safety**
Cookies	Very limited	Fixed	Can be seen in the URL and will be sent to the receiving server and back to the local computer.
Local storage	Big	Session or unlimited	Is stored on a local computer and nothing is sent to and from a server.
lawnchair	Big	Depends on the chosen technique	Is stored on a local computer and nothing is sent to and from a server.

In short, local storage is the newer way to save data locally. You can still use cookies, but the new privacy rules dictate that you must ask for permission before using them.

Summing up the complete data storage concept, we conclude that:

- The lawnchair application is a code package that is freely downloadable and can handle all your client-side storage needs. It can save using several methods, such as permanent DOM storage or Webkit-SQLite.
- The recommended downloadable code package is located at `https://github.com/jmo84/Lawnchair-plugin-for-ImpactJS`, since it comes with an ImpactJS plugin.
- Making use of the lawnchair storage system constitutes including the library and initializing one of our `GameInfo` class' variables as a lawnchair application's object. We can then store and retrieve data through the use of the `this` object since it inherits all of the lawnchair methods.

Extra functionalities for the RPG

In this section we will look at a few extra functionalities that might be particularly useful for a top-down game such as the **RPG** we designed in *Chapter 3, Let's Build a Role Playing Game*. First, we will implement character movement by mouse click, which is especially useful for mobile games, since touching the screen is the mobile equivalent of clicking a mouse. Then we will add a smart spawn point. This spawn point first checks whether spawning an entity would cause collision and adjusts its spawn coordinates accordingly. The third element adds conversation between the player and a non-playable character (NPC). The final add-on is a basic head-up display (HUD), which allows the players to keep track of their health.

Moving the player with a mouse click

Up until here, we moved our player around by use of the keyboard arrow keys. This is very intuitive, yet sometimes impossible. If you were to open your game on an iPad or any other mobile device, there is no way to move your character since no arrow keys are present. Here it would be more useful if our character would just walk towards the spot we touch on our screen. In ImpactJS, the mouse click and a touch are treated as the same thing, it just depends on the device. Therefore, implementing movement by a mouse click automatically results in movement by a touch for your mobile device. To make the player move around by clicking the mouse or touching the screen, you need to follow these steps:

1. In the `main.js` file, bind the mouse click to the action called `'mouseclick'`.

```
ig.input.bind(ig.KEY.MOUSE1, 'mouseclick');
```

2. Open the `player.js` file and add a few extra initial variables. We will need this once we start using the movement by a mouse function we are about to add.

```
name: "player",
movementspeed : 100,
mousewalking : 0,
takemouseinput : 0,
animSheet: new ig.AnimationSheet
|( 'media/player.png', 32, 48 ),
```

3. If the `movementspeed` variable was not already there as a `"player"` property, be sure to add it now. The `mousewalking` command is a flag variable; a value of `1` means the player must walk as commanded by a mouse click. The `takemouseinput` variable's value is set to `1` when the mouse is clicked and immediately returned to `0` after the goal coordinates are calculated. Without this variable, it would be possible to steer your character by the position of your mouse instead of a single click. This is a matter of choice; steering by mouse position instead of mouse click can certainly be part of a valid and intuitive control scheme.
4. Add the `mousemovement()` method to the `"player"` entity using the following code:

```
mousemovement: function(player){
if (player.mousewalking == 1 && player.takemouseinput == 1){
  player.destinationx = ig.input.mouse.x + ig.game.screen.x;
  player.destinationy = ig.input.mouse.y + ig.game.screen.y;
  player.takemouseinput = 0;
}
else if(player.mousewalking == 1){
  var distancetotargetx = player.destinationx - player.pos.x -
(player.size.x/2) ;
  var distancetotargety = player.destinationy - player.pos.y -
(player.size.y/2) ;
  if (Math.abs(distancetotargetx) > 5 ||
Math.abs(distancetotargety) > 5){
    if (Math.abs(distancetotargetx) >
Math.abs(distancetotargety)){
      if (distancetotargetx > 0){
        player.vel.x = player.movementspeed;
        var xydivision = distancetotargety / distancetotargetx;
        player.vel.y = xydivision * player.movementspeed;
        player.currentAnim = player.anims.right;
        player.lastpressed = 'right';
      }
      else{
        player.vel.x = -player.movementspeed;
        var xydivision = distancetotargety / 
Math.abs(distancetotargetx);
        player.vel.y = xydivision * player.movementspeed;
        player.currentAnim = player.anims.left;
        player.lastpressed = 'left';
      }
      }
    else{
      if (distancetotargety > 0){
        player.vel.y = player.movementspeed;
        var xydivision = distancetotargetx / distancetotargety;
        player.vel.x = xydivision * player.movementspeed;
        player.currentAnim = player.anims.down;
        player.lastpressed = 'down';
      }
      else{
        player.vel.y = -player.movementspeed;
        var xydivision = distancetotargetx / 
Math.abs(distancetotargety);
        player.vel.x = xydivision * player.movementspeed;
        player.currentAnim = player.anims.up;
        player.lastpressed = 'up';
      }
      }
    }
  else{
    player.vel.y = 0;
    player.vel.x = 0;
    player.mousewalking = 0;
    player.currentAnim = player.anims.idle;
  }
}
},
```

5. The length of this function might be a little daunting but actually the same logic is repeated several times. The function basically does two things: it can set destination coordinates and it can make the player move towards a target. In most cases it will not be necessary to calculate a new target. Thus, the first check is whether a new destination needs to be used. For this, both `player.takemouseinput` and `player.mousewalking` variables need to be `true`. In the calculation of the target location coordinates, a correction is applied for the position of the game screen.

6. Then the function proceeds with the actual movement; whether it should proceed or not is set by the value of the `player.mousewalking` variable (`True` or `False`).
7. If the player needs to walk, the actual distance to the target is calculated for both the x and y axes and stored in the local variables, `distancetotargetx` and `distancetotargety`. When the target is within 5 pixels of the player on either axis, the player will not move.
8. However, if the distance is greater than 5 pixels, the player will move towards his target in a linear manner. To make sure the player moves at the preset movement speed, it will do so at the axis on which the remaining distance is greatest. Let's say the player is far away from his target on the x axis but not so far on the y axis. In this case he will move at the preset movement speed on the x axis but less than the preset movement speed for the y axis. Also he will be facing left or right and not up or down.
9. The two most important triggering variables: `player.mousewalking` and `player.takemouseinput` are initialized at a value of `0`; they need to be put to a value of `1` when a mouse click is registered. This we do in the `update()` function as shown in the following code:

   ```
   if( ig.input.pressed('mouseclick')){
   this.mousewalking = 1;
   this.takemouseinput = 1;
   }
   ```

10. We just made sure that the game checks whether the mouse was clicked at every new frame.
11. If we now call our update function by adding a call to the `mousemovement()` method, the player will walk wherever a mouse click is registered on the screen.

    ```
    mousemovement();
    ```

12. Of course, our keyboard controls are still there and this will cause problems. In order to make both control methods work, all we need to do is set the value of the `player.mousewalking` variable to `0` every time one of the keys is pressed, as shown in the following code for the up arrow key:

    ```
    if(ig.input.state('up')){
      this.mousewalking = 0;
      this.vel.y =this.movementspeed;
      this.currentAnim = this.anims.up;
      this.lastpressed = 'up';
    }
    ```

13. Whether the value of the `player.mousewalking` variable is `0` or not needs to be checked constantly using the following code. If not, our old control system will immediately stop the movement because no keyboard input was registered.

    ```
    Elseif(this.mousewalking == 0){
      this.vel.y = 0;
      this.vel.x = 0;
      this.currentAnim = this.anims.idle;
    }
    ```

14. Finally, save your files and reload the game.

You should now be able to run around by clicking the mouse anywhere on the screen. You might notice slight course adjustments if the player encounters an obstacle. However, he is not smart enough to actually go around it if the obstacle is too big. As a player, you need to steer clear of obstacles yourself.

Let's have a look at how to create an intelligent spawn location. But before we do so, let's recap what was just covered:

- Being able to move the player with a mouse click is an interesting feature when moving to mobile devices, since there the keyboard is not an option. With ImpactJS, the click of a mouse is considered the same as touching the screen of an iPad.
- Currently our player can move around with the keyboard so we need to implement the possibility to use both the keyboard arrow keys and the mouse. All adjustments for this will take place within the player entity.
- We introduced a new method called `mousemovement()`, which is repeatedly called in the player's `update` function. At all times, our method will check whether the command for moving by a mouse click was given, and if so will move the player to the desired location.
- In addition to adding this new method, we needed to adjust our old movement code so it would allow combining both the arrow keys and the newly implemented movement by a mouse click.

Adding intelligent spawn locations

While building a level in the Weltmeister, you can immediately add your hostile entities to the level itself. This is nice, but sometimes adding a bit of unpredictability will increase the replay value of your game. This can be done by adding intelligent spawning: spawning enemies on random places but taking collision with other entities and the collision layer into account. In order to do this, we will need to create a new plugin with the following steps:

1. Create a new file and save it as `spawnlocations.js`.
2. Add the `'plugins.functions.spawnlocations'` command to your `main.js` file.
3. Create an `ig.spawnlocations` variable as an extension of the ImpactJS class, as shown in the following code:

   ```
   ig.module('plugins.functions.spawnlocations').
   defines(function(){
     ig.spawnlocations = ig.Class.extend({
     });
   })
   ```

4. Add the `spawnIf()` method, which is a callback function as shown in the following code. It can call itself again when certain conditions are met.

   ```
   spawnIf: function(x, y)
   {
     if (this.CollisionAt(x,y) || this.getEntitiesAt(x,y)){
       var x1 = x + Math.round(Math.random())*10;
       var x2 = x + Math.round(Math.random())*10;
       this.spawnIf(x1,x2); //recursion
     }
     ig.game.spawnEntity('EntityEnemy', x, y);
   },
   ```

5. The `spawnIf()` function takes in an x and y start coordinate and checks if there is collision with the collision layer or an entity. If this is the case, the original coordinates are adjusted with a random number of pixels on both axes. These new coordinates are then resubmitted to the `spawnIf()` function until it finds a free spot. Once no more collision is detected, an enemy is spawned on that location. The `CollisionAt()` and `getEntitiesAt()` functions that it requires are also part of the `spawnlocations` class.

6. The `getEntitiesAt()` function will detect entities that would be overlapping with the enemy that needs to spawn. The following code depicts the detection process applied by the `getEntitiesAt()` function:

```
getEntitiesAt: function(x, y)
{
  var n = ig.game.entities.length;
  var ents = [];
  for (var i=0; i<n; i++)
  {
    var ent = ig.game.entities[i],
    x0 = ent.pos.x,
    x1 = x0 + ent.size.x,
    y0 = ent.pos.y,
    y1 = y0 + ent.size.y;
    if (x0 <= x && x1 > x && y0 <= y && y1 > y)
      return true;
  }
  return false;
},
```

7. One by one, the entities are checked to see whether they overlap using their position, width, and height. If there is an overlap with a single entity, the loop is aborted and the `getEntitiesAt()` function returns the value `true`. If no overlap was detected it will return the value `false`.
8. While the `getEntitiesAt()` function checks for a possible collision with other entities, the `CollisionAt()` function checks if the enemy would overlap with the collision layer, as shown in the following code snippet:

```
CollisionAt: function(x,y)
{
  var Map = ig.game.collisionMap;
  var ent = new EntityEnemy();
  var res = Map.trace( x, y, x+ ent.size.x,y + ent.size.y,
ent.size.x,ent.size.y ); // position, distance, size
  // true if there is a collision on either x or y axis
  return res.collision.x || res.collision.y;
}
```

9. The most important function is the `collisionMap` method's `trace()` function. The `trace()` function will check if something is between the value of the `x` coordinate and the sum of the values of the `x` and `ent.size.x` variable's coordinate or between the value of the `y` coordinate and the sum of the values of the `y` and `ent.size.y` variable's coordinate. The last two arguments are the `size` of the entity. This is normally used to check a trajectory but we are using it to check a specific location. If there is a collision on either the x or y axes, the `CollisionAt()` function will return the value `true`, and the `spawnIf()` function will need to look for a new spawn location.
10. The last thing we need to do is actually spawn an enemy. This we can do within `MyGame` in the `main.js` file using the following code:
    ```
    var spaw = new ig.spawnlocations();
    spaw.spawnIf(100,200);
    ```
11. An enemy will now spawn at these coordinates if a free space is available, otherwise, the coordinates will be adjusted until a suitable spot is found.

Now that we added an intelligent spawn point to our game, it is time to move on to a somewhat complex game element: conversation. However, before we start the conversation process, let's quickly go through what we just did:

- The purpose of the intelligent spawn point is to find an open space for an enemy to spawn. For this, a check with both the entities already in the game and the collision layer of the level will be needed.
- We build a plugin that consists of three parts:
    - A callback function that will adjust the coordinates until a good location is found, and subsequently spawn the enemy. It makes use of the other two functions in our spawn point class.
    - The function that must check for potential overlap with other entities.
    - A function that checks for an overlap with the collision layer.
- Adding an enemy to the game can now be done by initiating a new spawn point and using its `spawnIf()` method to put a new enemy into the gaming world.

Introducing basic conversation

Many role playing games (RPGs) have conversations between the player and some of the non-playable characters (NPCs). In this section we will address a way to get a simple conversation into your game. The main prerequisite is the conversation data we added to our game earlier in this chapter. We need to build a conversation menu containing items that can be selected by the player using the following steps. Talkie, our lovely NPC from *Chapter 3, Let's Build a Role Playing Game*, will serve as our partner in crime. The player will not only have several answering options every time Talkie says something, but will also have the NPC react to what the player wants to say by opening new options. The cycle should be able to go on until all options are exhausted or the conversation is aborted abruptly:

1. Open a new file and save it as `menu.js` in the `conversation` subfolder of the `plugins` folder.
2. Add a `'plugins.conversation.menu'` command to your `main.js` file.
3. Create a `window.Menu` class as an extension of the ImpactJS class as shown in the following code:

```
ig.module(
  'plugins.conversation.menu'
)
.defines(function(){
  window.Menu = ig.Class.extend({
    init: function(_font,_choice_spacing,_choices,_entity){
      this.selectedChoice = 0;
      this.cursorLeft = ">>";
      this.cursorRight = "<<";
      this.cursorLeftWidth =
_font.widthForString(this.cursorLeft);
      this.cursorRightWidth =
_font.widthForString(this.cursorRight);
      var i,labeled_choice;
      for(i=0;i<_choices.length;i++){
        _choices[i].labelWidth =
_font.widthForString(_choices[i].label);
      }
      this.font = _font;
      this.choices = _choices;
      this.choice_spacing = _choice_spacing;
      this.entity = _entity;
      this.MenubackgroundMenubackground = new
ig.Image('media/black_square.png');
```

```
      this.Menubackground.height = this.choices.length *
this.choice_spacing;
    }
  }
},
```

4. Our menu `init()` function will require four input variables; we will turn all of them into `menu` properties so they are available in our `menu` methods; those four input variables are as follows:
    - `_font`: This is the font that we will be using
    - `_choice_spacing`: This is the space we want between each of the choices that will be shown on the screen
    - `_choices`: This is the array of choices the player has at a specific part of the conversation
    - `_entity`: This is the NPC that needs to talk to the player; in this case that would be `Talkie`
5. Our `init()` method contains some other important variables as follows:
    - `this.selectedChoice`: This is the variable that will store the array index of the currently selected choice. It is initialized at a value `0`, which is always the first element of any array and, thus, the first option of the player. The `this.selectedChoice` variable is important because the symbols `<<` and `>>` will be shown at the sides of the currently selected option as a visual aid.
    - `this.cursorLeft` and `this.cursorRight`: They are the variables that store the visual aid symbols `<<` and `>>`.
    - `this.cursorLeftWidth` and `this.cursorRightWidth`: They are the variables that store the length for the `<<` and `>>` symbols for the chosen font so that this can be taken into account when actually drawing the choices on screen.
    - `_choices[i].labelWidth`: This local variable stores the width that has been calculated for each of the choices. The calculated width is then stored in the menu property array `choices[i].labelWidth`. The `cursorLeftWidth` and `cursorRightWidth` variables will be used to determine the screen positioning while drawing the options on screen.

  - `this.Menubackground`: This variable will hold a black square, which serves as a background so that the white characters of the conversation are always readable, regardless of what the level currently looks like. The background adapts itself to the length of the longest option and the number of options. This way it doesn't take more space than is absolutely necessary.

6. The `draw()` method holds all of the menu logic, so we will discuss it in chunks using the following code:

```
draw: function(_baseX, _baseY){
  var _choices = this.choices;
  var _font = this.font;
  var i,choice,x,y;
  if (this.choices.length > 0){
    var Menubackground = new
ig.Image('media/black_square.png');
    Menubackground.height = this.choices.length *
this.choice_spacing;
    Menubackground.width = 1;
    for(var k=0;k<_choices.length;k++){
      choice = _choices[k];
      if(this.font.widthForString(choice.label)
>Menubackground.width){
        Menubackground.width =
this.font.widthForString(choice.label);
      }
    }
  Menubackground.width = this.Menubackground.width +
this.cursorLeftWidth + this.cursorRightWidth + 16;
  Menubackground.draw(_baseX-this.Menubackground.width/2,
_baseY);
  };
}
```

7. The first major thing the `draw()` function does is it adjusts the background of the menu so that it is always big enough to fit the different sentences, given the chosen font. This logic, among others, could just as well have been stored in an `update()` function instead of a `draw()` function. This is a matter of choice and you are of course free to rewrite the `menu` class as you please. The bottom line is that both the `draw()` and `update()` functions are called in every frame, which is the most important common attribute of these methods. In the following code we can check out the functionality of a `draw()` function:

```
for(i=0;i<_choices.length;i++){
  choice = _choices[i];
  choice.labelWidth = _font.widthForString(choice.label);
  y = _baseY + i * this.choice_spacing + 2;
  _font.draw(choice.label, _baseX, y,
ig.Font.ALIGN.CENTER);
  if (this.selectedChoice === i){
    x = _baseX - (choice.labelWidth / 2) -
this.cursorLeftWidth - 8;
    _font.draw(this.cursorLeft, x, y - 1);
    x = _baseX + (choice.labelWidth / 2) + 8;
    _font.draw(this.cursorRight, x, y - 1);
  }
}
```

8. Now the position of the text is determined and every choice is written on the screen. A check is done to see which option is currently selected. This option is encapsulated by the **<<** and **>>** symbols to make the player aware of what he is about to choose. To add these functionalities we will look into the following code:

```
if(ig.input.pressed('up')){
  this.selectedChoice--;
  this.selectedChoice = (this.selectedChoice < 0) ? 0 :
this.selectedChoice;
}
else if(ig.input.pressed('down')){
  this.selectedChoice++;
  this.selectedChoice = (this.selectedChoice >=
_choices.length) ?
_choices.length-1 : this.selectedChoice;
}
else if(ig.input.pressed('interact')){
  var chosen_reply_key = _choices[this.selectedChoice].npcreply();
  ig.game.spawnEntity('EntityTextballoon',this.entity.pos.x -
10,this.entity.pos.y - 70,
{wrapper:npc_con.NPC_SPEECH[chosen_reply_key]});
  this.choices =_choices[this.selectedChoice]
.changechoices(chosen_reply_key);
}
```

9. The player has three options: he can press the up arrow, down arrow, or the interact button on the keyboard; the last action state corresponds to the *Enter* key. Here we will explain how it is done for a regular desktop. It is a good exercise to try implementing this for mobile devices:
   - If the `'up'` input state is activated, the `'up'` state should currently be bound to the up arrow of the keyboard and the selected choice shifts one position upwards. In the array this translates to an element with a lower index. However, if the position 0 within the index is reached, it can't go any lower since this is the first option. In this case it stays at the first option.
   - An equivalent logic is used for going down the menu with the down arrow key.
   - If the `'interact'` state is not yet bound to the *Enter* key, do so now by adding the `ig.input.bind( ig.KEY.ENTER, 'interact' );` command to the `main.js` file. The player makes his choice by pressing the *Enter* key. Using the `npcreply()` function, the NPC knows what to say and will spawn a text balloon containing his reply. Based on this reply, the `this.choices` function will be filled with new choices for the player to pick from.

10. The menu is composed of different items; every separate option corresponds to a single menu item. Add this menu item class to the `menu.js` file using the following code:

```
window.MenuItem = ig.Class.extend({
  init: function(label,NPC_Response){
    this.label = label;
    this.NPC_Response = NPC_Response;
    this.entity = entity;
    },
  });
});
```

11. A menu item is initialized with the following two input parameters:
    - The label, which is the actual text of a choice or option.
    - `NPC_Response`, which is the Primary Key of an NPC reply. With this key it is possible to look up what the NPC needs to answer and construct new options for the player to choose from.

12. The `npcreply()` method uses the `NPC_Response` key, as shown in the following code, to look up the array number of the reply that the NPC will give in the `NPC_CON` array we constructed earlier in this chapter:

```
npcreply: function(){
  for(var i= 0;i<=npc_con.NPC_CONVO_KEY.length; i++){
    if (npc_con.NPC_CONVO_KEY[i] == this.NPC_Response){
    return i;
    }
  }
},
```

13. As you might recall, there are only two arrays that together make up our entire conversation:
    - `NPC_CON`: This array constitutes everything our NPC has to say
    - `PC_CON`: This array constitutes everything our player can say

14. In the menu code, the key is stored in a local variable called `chosen_reply_key` and is then re-used in the following two ways:
    - To make the NPC reply
    - To construct new options by inputting it as an argument to the `changechoices()` method

15. Finally, the `changechoices()` method takes in what the NPC has said, as shown in the following code, and constructs new options by going through the `PC_CON` array we built earlier in this chapter.

```
changechoices: function(chosen_reply_key){
  var choices =  []
  for(var k= 0;k<=pc_con.REPLY_SET_KEY.length; k++){
    if (pc_con.REPLY_SET_KEY[k] ==
npc_con.REPLY_SET_KEY[chosen_reply_key]){
      choices.push(new MenuItem(pc_con.PC_SPEECH[k],
pc_con.NPC_CONVO_KEY[k]));
    }
  }
return choices;
}
```

The conversation is a loop, which theoretically can go on forever. However, we still need a start. We could do this by initializing our `Talkie` NPC menu with a few options in the `Talkie` NPC itself. This is a very pragmatic approach, but then again, so is the entire implementation of this conversation plugin, and you are free to adapt and extend it as you please.

We still need to adjust our `Talkie` entity before we can start talking to him:

1. Open the `talkie.js` file and add the following code to the file as properties:

```
var i;
this.choices = [
new MenuItem(pc_con.PC_SPEECH[0],
pc_con.NPC_CONVO_KEY[0],this),
new MenuItem(pc_con.PC_SPEECH[1],
pc_con.NPC_CONVO_KEY[1],this),
new MenuItem(pc_con.PC_SPEECH[2],
pc_con.NPC_CONVO_KEY[2],this)
];
var menufont = new ig.Font('media/04b03.font.png');
this.contextMenu = new Menu(menufont,8,this.choices,this);
```

2. We have now added a conversation menu for Talkie and initialized it at the first three options of our `PC_CON` array.
3. Now we need a function that checks whether Talkie was actually selected. Otherwise, there will be conflicts if we introduce several NPCs at the same time. To check if Talkie has been actually selected we write the following code:

```
checkSelection:function(){
  this.mousecorrectedx = ig.input.mouse.x + ig.game.screen.x;
  this.mousecorrectedy = ig.input.mouse.y + ig.game.screen.y;
  return (
    (this.mousecorrectedx >= this.pos.x && this.mousecorrectedx <=
this.pos.x+this.animSheet.width)
&& (this.mousecorrectedy >= this.pos.y && this.mousecorrectedy <=
this.pos.y+this.animSheet.height)
    );
  },
}
```

4. The function will check where the mouse was clicked and correct its position for the game screen. No correction would be necessary if our level would perfectly fit into the viewport, but this is almost never the case, so a correction needs to take place. The function returns a `true` or `false` value. It returns the value `true` if the entity was selected, and returns `false` if it wasn't.

5. In our `update()` method, we can now check for a mouse click and see if Talkie was actually selected using the following code:

   ```
   if( ig.input.pressed('mouseclick') ) {
     this.contexted = this.checkSelection();
   }
   ```

6. If it was, we set its brand new property `contexted` as `true`. In case Talkie wasn't selected, `contexted` is set to `false`.
7. If the `Talkie` entity was clicked and there is a menu available, it will be drawn underneath the `Talkie` entity with the following code:

   ```
   draw: function() {
     if(this.contexted && this.contextMenu){
       this.contextMenu.draw(this.pos.x+(this.animSheet.width/2)-
   ig.game.screen.x,this.pos.y+(this.animSheet.height)-
   ig.game.screen.y);
     }
   this.parent();
   },
   ```

8. Talkie is now ready to talk! Certainly try setting up your own conversation and see it unfold in your game.

As a last useful plugin before we move on to discussing some advanced AI, we will add a nice bar, visually presenting the player's health. However, before we do so, we will first recap the conversation plugin:

- We want to set up a conversation between our player and an NPC. For this, we will make use of the data we imported earlier in this chapter and a new plugin called `Menu`.
- The `Menu` plugin consists of two parts: the menu itself and the options within the menu. We create both as an extension of the `ImpactJS` class.
- After setting up both the `Menu` plugin and the menu items, some extra adjustments need to be made to our friendly NPC Talkie. When the player clicks the `Talkie` entity with the mouse button, the menu, featuring several options, should appear underneath him. When one of the options is chosen, Talkie replies. To show the reply, we make use of the speech balloon, which we created in *Chapter 3, Let's Build a Role Playing Game*.
- The entire conversation is a loop, which ends when either the player or the NPC runs out of sentences or the player just walks away.

Adding a basic Head-Up Display

Our player has health, but he is not aware of how much or little he is left with at any given time. Because being aware of how much health you have left as a player is so vital, we are going to show this on the screen as both a number and a health bar. For this, we build ourselves the HUD plugin using the following steps:

1. Open a new file and save it as `hud.js` under a `hud` subfolder of the `plugin` folder.
2. Add the `'plugins.hud.hud'` command to the `main.js` script.
3. Start off by inserting the following code in the new `plugin` file:

```
ig.module('plugins.hud.hud').
defines(function(){
  ig.hud = ig.Class.extend({
    canvas  : document.getElementById('canvas'), //get the canvas
    context : canvas.getContext('2d'),
    maxHealth  : null,
    init: function(){
      ig.Game.inject({
        draw: function(){
          this.parent();
          // draw hud if there is a player
          if(ig.game.getEntitiesByType('EntityPlayer').length  !=
0){
            if (this.hud){
            this.hud.number();
            this.hud.bar();
            }
          }
        }
      })
    },
  }
}
```

4. As usual, we define a new class based on the ImpactJS class. We initiate two variables: canvas and context, which will allow us to see whether the game is being viewed. Also, we initiate a `maxHealth` variable at a value `null`. However, unlike the usual condition, we use the inject technique as we did when we constructed our debug panel. When extending code, you create a new instance of that original code and supply it with a new name. It is, by all means, a copy of the original with the difference being the extra code you added. But while injecting, you modify the original code. In this case we overwrite the `draw()` function of our game. The `this.parent()` function points to our former `draw()` function, so everything which was already there is kept. What we add is the check for the presence of a player entity. If a player is in the game, a HUD is drawn. Our HUD is comprised of two parts: the number and the health bar.

5. The `number` function will draw a black and slightly transparent rectangle in which the health will be visible, using the following code:

```
number: function(){
  if(!this.context) return null;
  var player =
ig.game.getEntitiesByType('EntityPlayer')[0];
  // draw a transparant black rectangle
  var context = this.canvas.getContext('2d');
  context.fillStyle = "rgb(0,0,0)";
  context.setAlpha(0.7); //set transparency
  context.fillRect(10,10,100,30);
  //draw text on top of the rectangle
  context.fillStyle = "rgb(255,255,255)";
  context.font = "15px Arial";
  context.fillText('health: ' + player.health,20,30);
  //font used is the default canvas font
  context.setAlpha(1);
  return null;
},
```

6. In the first part of our `number()` function, we define and draw the rectangle. Since it will need to be situated underneath the number, it needs to be drawn first. Unlike before, we are directly using canvas element's properties to draw on the screen. The font, for instance, does not need to be set by using the ImpactJS `ig.font` function. As shown here, you can just write characters to the screen by directly addressing the canvas and setting the canvas' `font` property. The canvas properties we use here are pretty straightforward and are listed as follows:

    ◦ `fillstyle`: This property will set the color

- `font`: This property will set the font
- `setAlpha()`: This property will set the transparency, with a value `1` being solid and `0` being fully transparent
- `fillRect()`: This property will draw a rectangle to the screen at a given position with a given width and height
- `fillText()`: This property will draw text on the screen at a certain position

7. Our health bar function works in a similar way as that of the number function, as shown in the following code:

```
bar: function(){
  if(!this.context) return null;
  var player = ig.game.getEntitiesByType('EntityPlayer')[0];
  // draw a transparant black rectangle
  var h = 100*Math.min(player.health / this.maxHealth,100);
  var context = this.canvas.getContext('2d');
  context.fillStyle = "rgb(0,0,0)";
  context.setAlpha(0.7);
  context.fillRect(10,50,100,10);
  //either draw a blue or red rectangle on top of the
black one var color = h < 30 ? "rgb(150,0,0)" :
"rgb(0,0,150)";
  context.fillStyle = color;
  context.setAlpha(0.9);
  context.fillRect(10,50,h,10);
  context.setAlpha(1);
  return null;
},
```

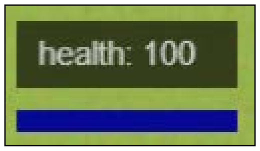
8. Here we draw two rectangles on top of each other. The bottom one is black at all times and is slightly transparent. The top rectangle is either blue or red, depending on how much health the player is left with. If the player's health value is `30` or higher, the bar will be blue, otherwise it will be red indicating impending demise.
9. The size of the black transparent bottom bar is always the same, but its width depends on how much health a player has when he starts the game. This we can capture with the `setMaxHealth()` method as depicted in the following code:

```
setMaxHealth: function(health){
  this.maxHealth = health;
}
```

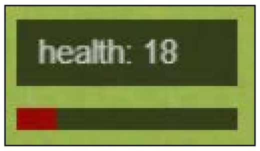
10. All we now need to do is initialize a HUD and feed it with the player's health using our `setMaxHealth()` method. Add the following code to the `main.js` file:

```
MyGame = ig.Game.extend({
  font: new ig.Font( 'media/04b03.font.png' ),
  ai: new ig.general_ai(),
  hud: new ig.hud(),
  init: function() {
    this.loadLevel(LevelLevel1);
    var player = ig.game.getEntitiesByType('EntityPlayer')[0];
    this.hud.setMaxHealth(player.health);
  }
}
```

11. While reloading the game, we should now have a blue health bar and an indication that we have a **health** value of **100** left, as shown in the following screenshot:



12. However, after a small battle with our foes we can see by our red health bar that it is time to consult a doctor, as shown in the following screenshot:



Now that we took a look at some interesting extensions to *Chapter 3*, *Let's Build a Role Playing Game*, let's revisit our AI and introduce a new level of complexity. Before moving on, let's quickly go over the way we constructed our HUD:

- A HUD or Head-Up Display offers a quick view on several key measures of the player, which help the player to be successful. In a shooter, this depicts how much ammo he is left with, both in total and in the current magazine. It can indicate other items or his overall score. Here we allow him to keep track of his health using a classic health bar.

- The `hud` plugin is an extension of the ImpactJS class and has two elements: the health as a number and as a colored bar. They both have their separate methods within the `hud` plugin. You can extend the `hud` plugin yourself by adding new methods that represent other trackable statistics.
- In building the HUD, we used the `canvas` properties as an alternative to using ImpactJS classes such as `ig.font`.

Artificial intelligence: The hive mind

In *Chapter 3, Let's Build a Role Playing Game*, we already covered AI and why behavior should be separated from the process of decision making. We have also taken a look at strategies, but only applied a single strategy: attacking. Here we will set up a supplementary layer of intelligence that will decide which entity will follow which strategy. Because the decision making process takes into account all enemies in the same level, we call it hive mind intelligence. It is very similar to the queen of a beehive or the general on a battlefield who decides who should attack and who should stay put. The strategy decided in our hive mind is sent to the AI that we put in place during *Chapter 3, Let's Build a Role Playing Game*, where it is interpreted and translated into behavior. The behavioral commands are in turn sent to the entity itself, which then acts upon them. Let's create our general `ai` plugin using the following steps:

1. Open a new file and save it as `general_ai.js`.
2. Insert the `'plugins.ai.general_ai'` class in the `main.js` file.
3. Create the `ig.general_ai` class as an ImpactJS class extension. Generally, the class `general_ai.js` has been created as shown in the following code:

```
ig.module('plugins.ai.general_ai').
defines(function(){
  ig.general_ai = ig.Class.extend({
    init: function(){
      ig.ai.STRATEGY = { Rest:0,Approach:1};
  },
}
```

4. The first thing we do is define the possible strategies. Here we will issue only two strategies: `Approach` or `Rest`.

5. The `getStrategy()` function is located where our hive mind decides to keep it, and it is the function that shall be called by our AI in order to receive a strategy. This strategy is in turn translated into behavior using the following code:

```
getStrategy: function(ent){
  // part 1: get player and list of enemies
  var playerList = ig.game.getEntitiesByType('EntityPlayer');
  var player = playerList[0];
  var EnemyList = ig.game.getEntitiesByType('EntityEnemy');
  // part 2: store distance to player if that enemy has enough
health to attack
  var distance =  [];
  for(var i = 0;i < EnemyList.length; i++){
    //for every enemy > 100 health: put in array
    EnemyList[i].health > 100 ?
distance.push(EnemyList[i].distanceTo(player)) : null;
  }
  // part 3: decide on strategy: attack or stay put?
  var Mindist = Math.min.apply(null,distance);
  var strategy = (ent.distanceTo(player)===Mindist ||
distance.length === 0) ? ig.ai.STRATEGY.Approach:
ig.ai.STRATEGY.Rest;
  return strategy;
}
```

6. The `getStrategy()` method contains our entire hive mind logic and consists of three main parts:

   ° First, a list of enemies and the player entity are each assigned to a local variable.

   ° These local variables are then used to calculate the distance between each enemy and the player for those enemies who have more than 100 health value. Every enemy below the health value of 100 is regarded to be weakened and too scared to attack. This code could be made more complex by adding personality to each enemy. We could, for example, initialize each enemy with a `courage` property, filled with a random number within the health range of our enemy; in our case this is `0` to `200`. This way we could decide whether a certain enemy feels bold enough to attack by comparing his current health with his courage instead of comparing with a fixed value. Certainly try this yourself; it adds depth and unpredictability to the game.

  - Finally, all enemies bold enough to attack are compared by their distance to the target, and only the closest of them will attack. The others are issued the `Rest` strategy and will only attack when they become the closest enemy around. As a player you should still be careful though. If none of them feel strong enough to attack on their own, they will join forces and attack together.

7. In our previously built AI, we now need to call the `getStrategy()` function using the following code:

   ```
   getAction: function(entity){
   this.entity = entity;
   if(ig.game.ai.getStrategy(entity) == ig.ai.STRATEGY.Approach){
   ```

8. If the strategy is `Approach`, the AI will translate this into the appropriate action.

   ```
   return this.doAction(ig.ai.ACTION.Rest);
   ```

9. If the strategy is something else, it is immediately translated into the `Rest` action. Because we only have these two strategies, this makes sense. If you have more strategies, you will need more checks.

Now that we have extended our AI to incorporate strategies, it is time to take a look at the final part of this chapter: implementing game analytics with Playtomic. Let's quickly recap the hive mind AI before moving on:

- The hive mind is an overhead decision making organ, which will issue strategies to the different entities in the game. It is a way to make them act as part of a group rather than a bunch of unorganized individuals.
- In *Chapter 3*, *Let's Build a Role Playing Game*, we had the decision-making process, which was translated into behavior. Now we have a strategy, which translates into individual decision making, which is in turn converted to behavior.
- The hive mind plugin is separate from our AI that we built in *Chapter 3*, *Let's Build a Role Playing Game*. This way we can still return to our individualistic AI with only minor code corrections.

- The hive mind logic follows three main steps:
    - Fetch all enemies and the player within the level.
    - Check the health value for each enemy to see if he is a viable candidate for an attack.
    - From this list of viable enemies, choose the one who is closest to the player and let him attack. How the enemy will perform this attack is not specified by the general AI; this is a decision of the individual AI.

Implementing Playtomic for game analytics

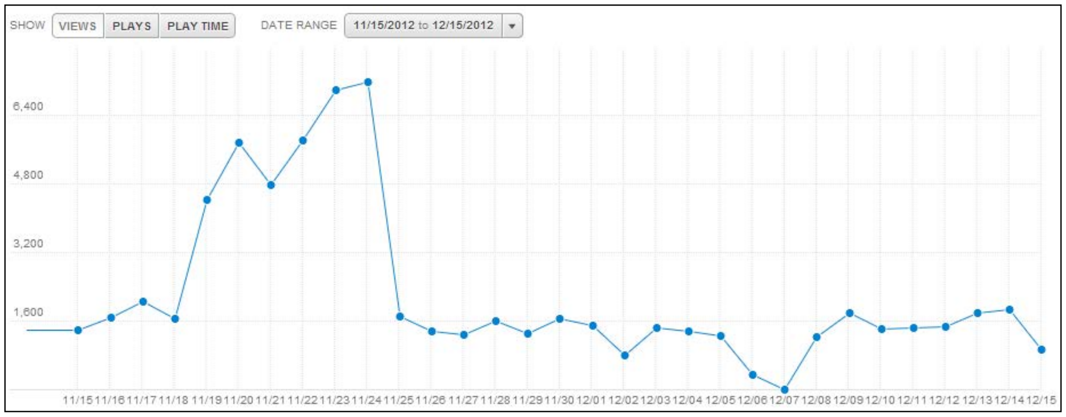
Playtomic can be regarded as the Google Analytics for games. You can tag certain parts of your game and check if they are used often or not. For example, if you have a hidden level in your game, you would be able to see how many times and by how many different gamers it was discovered by tagging the `loadlevel()` function of this hidden level. Then you would be able to determine whether it might be too easy or too hard to discover, and adjust your game accordingly. But this is just one of the many ways you can apply game statistics. However, you need to be aware that tagging your game will tax its performance to a certain degree. And so, tagging every inch of code might not be as fertile a solution as intended. In addition, you would be left with massive amounts of data that you would then need to analyze, which could be a daunting task.

In addition to supplying you with insights on your game usage, Playtomic allows you to store certain things on their server, such as scores, which you can turn into a leaderboard.

If all of this sounds good to you, by all means go over to `https://playtomic.com/` and create yourself a free account.

A few warnings are appropriate though. Playtomic is still in its infancy, and this translates into some bugs or illogical choices. For example, the default practice for saving a score in a leaderboard is not to overwrite the first one, even if the new one is higher. This doesn't make sense for a leaderboard and even the documentation indicates the default setting to be set on allowing score overwrites. Contacting the Playtomic server will slow down your game load, and often data is lost because no stable connection was made.

But even though there are flaws in the implementation, server speed, and documentation, Playtomic is worth taking a look at if you want to gather insights about your game. The following screenshot depicts the data collected by Playtomic and its representation:



In order to implement Playtomic, there are a few things you will need to do:

1. Create a Playtomic account and fetch your data transmission credentials. You will need these to set up a connection to their server.
2. In the `index.html` file we will need to include the Playtomic source script as shown in the following code. Certainly check what the latest version is, at the time of installation. At the time of writing this book it was Version 2.2 but these things develop fast.

```
<body>
  <canvas id="canvas"></canvas>
  <script type="text/javascript"
src="http://api.playtomic.com/js/playtomic.v2.2.min.js"></script>
</body>
```

3. Open a new file and save it as `PlayTomic.js` under the `data` subfolder in the `plugins` folder. Here we will place the functions that we will need in order to work with Playtomic.
4. Include this plugin file in our `main.js` script as shown in the following code line:

```
'plugins.data.PlayTomic'
```

5. Define the `PlayTomic` plugin module using the following code:

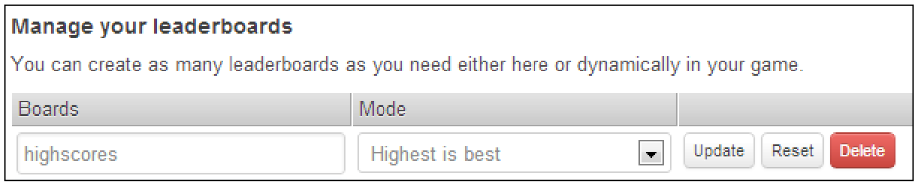
```
ig.module('plugins.data.PlayTomic').
defines(function(){
// module to store and retrieve things with Playtomic
ig.PlayTomic= ig.Class.extend({
userName : null,
success: true,
scores: null,
init: function(){
  ig.log('Trying to start Playtomic...');
  try{
    Playtomic.Log.View( 951388, 'b05b606fc66742b9',
'f41f965c47a14bcfa7adee84eff714', document.location );
    //your login credentials
    Playtomic.Log.Play();//game start
    ig.log('loading Playtomic success ...')//could connect
message
  }
  catch(e){
    this.success = false; //could not connect
    ig.log('Failed loading Playtomic ...')//could not
connect message
  }
},
```

6. Our new Playtomic class will have the task of saving the player's score on the Playtomic server. However, first a connection with the server needs to be established; this is done in the `init()` function. Inserting log messages at key moments is extremely useful when implementing and testing your Playtomic setup. You will need to fill out your own connection credentials in the highlighted part of the previous code.

7. Once we have a connection, we will need to send data. Since we are going to save the score, we need a `saveScore` method as shown in the following code:

```
saveScore: function(name, score1){
  var score = {Name: name, Points: score1};
  Playtomic.Leaderboards.Save
(score,'highscores',this.submitComplete,
{allowduplicates:true});
},
```

8. The `Playtomic` class has a **leaderboards** property to which you can save the player's score using its `save()` method. You will need to specify that you want to save to the high scores table and add the value of the score. You can name the table yourself in your **leaderboards** settings on the Playtomic website, as shown in the following screenshot:



Manage your leaderboards

You can create as many leaderboards as you need either here or dynamically in your game.

Boards	Mode	
highscores	Highest is best	

9. We add an optional function that will give us feedback if the submit was successful. Keeping track of all the sent and received data is highly recommended while working with Playtomic. As a final parameter, we will allow duplicates on the leaderboard so that one person can have several scores on the board.

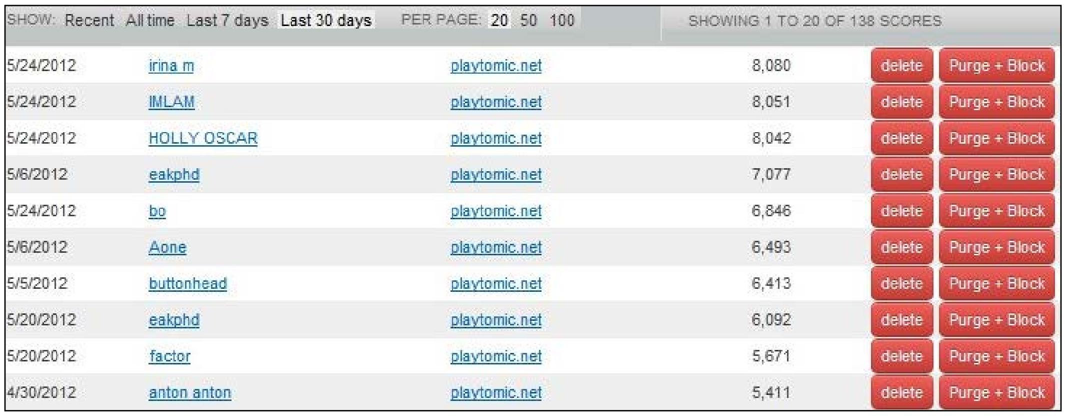
10. The `submitComplete()` function is just a way for us to keep track of whether a certain data transmission was successful or not:

```
submitComplete: function( response ){
  if( response.Success ){
    ig.log( 'Successfully Logged!' ); //submit success
    ig.log( response );
  }
  else{
    ig.log( 'Unable to Save High Score!' ); //submit fail
  }
},
```

11. All that is left to do now is integrating our `PlayTomic` analytics, as shown in the following code, using the `GameInfo.saveScore()` function we built for saving scores with the lawnchair application:

```
this.PlayTom = new ig.PlayTomic();
this.saveScore = function(){
  this.store.save({score:this.score});
  if(this.PlayTom.success){
    try{
    //service sometimes failes to load
      this.PlayTom.saveScore(this.userName,this.score);}
      catch(e){
        ig.log("Could not load to Playtomic");
      }
    }
  }
}
```

12. Our `saveScore()` method now not only saves the score locally by use of the lawnchair application, but also sends the result to the Playtomic server where it is put in a leaderboard, as shown in the following screenshot:



SHOW: Recent All time Last 7 days Last 30 days PER PAGE: 20 50 100 SHOWING 1 TO 20 OF 138 SCORES

5/24/2012	irina m	playtomic.net	8,080	delete	Purge + Block
5/24/2012	IMLAM	playtomic.net	8,051	delete	Purge + Block
5/24/2012	HOLLY OSCAR	playtomic.net	8,042	delete	Purge + Block
5/6/2012	eakphd	playtomic.net	7,077	delete	Purge + Block
5/24/2012	bo	playtomic.net	6,846	delete	Purge + Block
5/6/2012	Aone	playtomic.net	6,493	delete	Purge + Block
5/5/2012	buttonhead	playtomic.net	6,413	delete	Purge + Block
5/20/2012	eakphd	playtomic.net	6,092	delete	Purge + Block
5/20/2012	factor	playtomic.net	5,671	delete	Purge + Block
4/30/2012	anton anton	playtomic.net	5,411	delete	Purge + Block

There is a lot more to Playtomic than was covered here but that will be up to you to discover in full. With this humble introduction, you should already feel confident in starting up your own game analytics. Though, be aware that privacy regulations apply and they are constantly changing. It is best to ask the player's permission to keep the game statistics and make sure you take this into account while implementing your Playtomic code.

Summing up the complete process of introducing Playtomic in our game, we conclude that:

- Playtomic is the Google Analytics of mobile games and is free and relatively easy to implement.
- The first thing you need after creating a Playtomic account is a connection to their script, which can be included in your `index.html` file.
- A connection to the Playtomic server needs to be established. This is done using the credentials of your account, though you are free to test with the credentials in the example code.
- The goal of this introduction was to send scores from our game platform to the Playtomic server where they can be represented in a leaderboard. For this we made our own Playtomic plugin.

Summary

In this chapter we have taken a look at some of the more advanced things you can do with your game, and applied them on the RPG that we designed in *Chapter 3*, *Let's Build a Role Playing Game*.

We constructed an introductory, Victory, and Game-over screen and made our game prompt for the player's name so we could show it on the intro screen.

We went deeper into how to debug code by unit testing and made our own ImpactJS debug panel. Then we took a look at handling data and ways to store it on the player's device. The RPG was extended with a few fun elements, such as a way to move the player by a mouse click, an intelligent spawnpoint, NPC conversation, and a health bar.

Our AI was enhanced by introducing overhead strategy decision making, such as the hive mind. Finally we took a look at Playtomic and how to send the player scores to the Playtomic database.

In the next chapter we will have a look at music and sound effects. The aim is to get the basic sounds and music needed to start making your first game.

6
Music and Sound Effects

The music and sound effects are like the cherry on a cake: they can greatly improve the game experience when implemented correctly, but if not, at least you still have the cake. The big-budget games are nowadays always accompanied by original and exquisite songs and tunes. The game music niche has grown in the past decades, and there are many composers out there who are dedicated to making game music.

The following is a list of a few such great composers:

- Koji Kondo (Mario and Zelda series)
- Nobuo Uematsu (Final Fantasy series)
- Masato Nakamura (Sonic, Metal Gear Solid, and Metroid Prime series)
- Michael Giacchino (Call of Duty, Medal of Honor: Allied Assault)
- Bill Brown (Command and Conquer Generals, Enemy Territory, Rainbow Six)
- Jeremy Soule (Elder Scroll series, Star Wars Old Republic, Total Annihilation, Neverwinter Nights, Baldur's Gate, Guild Wars, Company Of Heroes, Putt-Putt)

These people certainly know how to make mind-blowing music that gives an incredible added value to the game experience. The tunes used in these games often become as iconic and memorable as the games themselves. If you watch a very old movie, you will notice that they used a lot less music and sound effects than they do nowadays and this makes them practically unwatchable to many of us today. Try stripping all the background music from any recent movie and you might find it dull to watch, even though the story stays the same. The same goes for many games, especially for adventure games, where well-composed background music is very important because it helps drag you into the storyline.

Similarly, horror games without sound effects and threatening music are almost unthinkable. Tunes and sound effects are vitally important to set the mood for a scene. A good example is the celebrated game **Resident Evil**. In this zombie game, even if nothing happens for 20 minutes, you will still be on the edge at all times. It's the sounds and the threatening music that makes you instinctively hesitant to open the next door. So, before choosing music and sound effects, think about the feeling you want your gamers to have when playing. For invoking feelings, nothing is more influential than perfectly picked music and sounds.

In this chapter we will have a look at a few sources for game music, apart from these rather expensive composers. We will have a quick look at FL Studio, which can be used to compose your own music. Finally we shall integrate the music and sound effects in ImpactJS.

Making or buying the music

In case you decide that you want to add some music to your game, it is still a question of either making it or buying it. It seems that as a 2D game builder, you need to know a little bit of everything: you must be able to understand game psychology, actually program your game, make graphics for it, and even compose its music. Sounds like you need to be a one-man army to pull off such a feat. However, educating yourself in the fields of graphic design and music might be a waste of time. Although being a generalist is a great characteristic, consider how much time it would cost for you to compose music for your games, as opposed to buying it from someone else.

In this chapter both options are supported. First, let's have a look at some websites that can provide you with music and sound effects.

Buying tunes and sound effects

If you need some **music** for your game, you could hire a personal composer like Jeremy Soule. However, assuming you don't have a multimillion-dollar budget, the following websites can be of use:

`www.craze.se`

On *Craze*, a wide repertoire of music can be found. The songs can be listened to in advance and the prices range from $15 to $60 per track. They can also be bought as packs, which significantly reduces the overall cost.

If you are looking for a somewhat more affordable supplier, you can take a look at *Lucky Lion Studios* at the following link:

`www.luckylionstudios.com`

Most tracks are sold at $5. They accept custom commissions and will even make a difference between buying exclusive or non-exclusive rights to the custom project, allowing you to cut costs on a custom assignment.

Finally, if you are looking for some free music there is *Nosoapradio*, which can be found at the following link:

`www.nosoapradio.us`

This website has it all; you can listen to and download over 300 tracks (over 12 hours of music) and it is all free to use as you please. The website even provides the tracker to a torrent file for you to download the 1 gigabyte of music all at once. This is a great site in case you wish to have some music as a placeholder or even release an actual game with it.

There are also a few sites out there for buying sound effects:

- *Pro sound effects* allows you to buy from a wide array of different sounds for $5 per effect at the following link:

  `www.prosoundeffects.com`

  You can also buy entire libraries on a certain topic, such as animal sounds, for example. The prices for these packages can range anywhere from $40 to $15,000.

- *Radish patch* does custom work for $45 per hour but also sells premade sound effects for $8 or $80, depending on what you plan to do with it. It can be found at the following link:

  `www.radish-patch.com`

  If you plan on selling over 5000 copies of your game, they charge $80 instead of $8 for each sound effect you use.

- The list wouldn't be complete without a free site for you to plunder, which can be found at the following link:

  `www.mediacollege.com/downloads/sound-effects/`

  Media college provides a reasonable number of free sound effects over a wide array of topics. The only thing they ask in return is a donation in case you like what they offer.

Sound effects, unlike good music, aren't too difficult to make. All you need is a list of sounds you require, a decent sound recorder, and a little bit of spare time (and maybe a few crazy friends to help you produce them). So when it comes to deciding whether you have to make or buy sound effects, making them yourself is recommended, except if you need some really exceptional-quality effects.

Let's now have a look at the basics of FL Studio for our music.

Making a basic tune using FL Studio

FL Studio is a digital audio workstation, formerly known as FruityLoops. Following is the logo of FL Studio:

FL Studio is not a freeware but a demo that can be downloaded at their website:

`www.fl-studio.en.softonic.com`

FL Studio is regarded as the most complete virtual studio that is currently out there. However, FL Studio is not yet available for Linux.

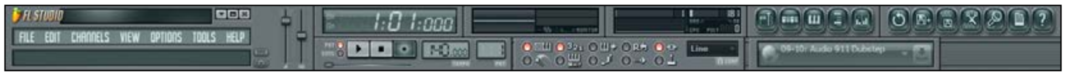
For Linux users, **LMMS** can be a good (and free) though less powerful alternative. Following is the logo of LMMS:

You can download LMMS from the following link:

`http://lmms.sourceforge.net/download.php`

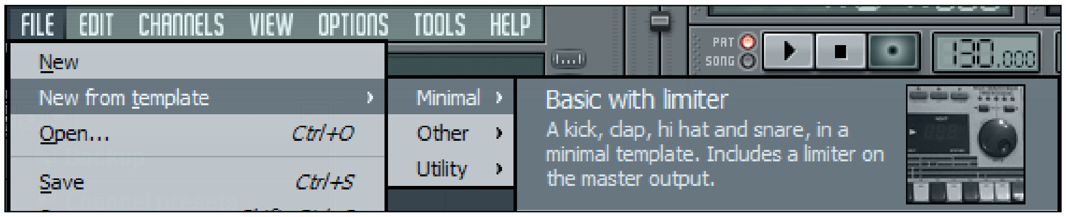
Since the aim of this book is not to give an in-depth insight into music development, only the basics of FL Studio are covered here.

When opening FL Studio, the first elements to notice are the top menu bars as shown in the following screenshot:



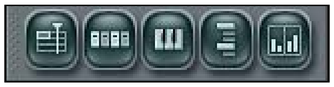
We can roughly distinguish between three main parts. On the left-hand side are all the menus you can expect any program to have these days: **FILE**, **TOOLS**, **VIEW**, **OPTIONS**, and so on. The middle of the bar provides quick access to the play, stop, and other buttons directly related to the song you are working on. On the right-hand side we find some quick-access buttons for the important elements of FL Studio.

When creating a new file, FL Studio allows you to start from a template, which is great for beginners.



Basic with limiter
A kick, clap, hi hat and snare, in a minimal template. Includes a limiter on the master output.

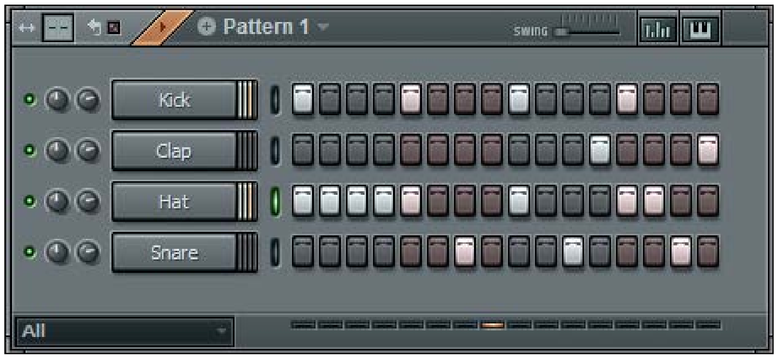
For instance, **Basic with limiter** will immediately provide the user with the different elements of a drumline. That way you don't need to figure out the basic components yourself. The quick-access buttons to the five most important elements of FL Studio are from left to right: the playlist, step sequencer, piano roll, file browser, and the mixer as shown in the following screenshot:



If you open the step sequencer, you will notice that your first sequence, called **Pattern 1**, has four elements predefined: **Kick**, **Clap**, **Hat**, and **Snare**. These four form the basis to your drumline as explained in the following list:

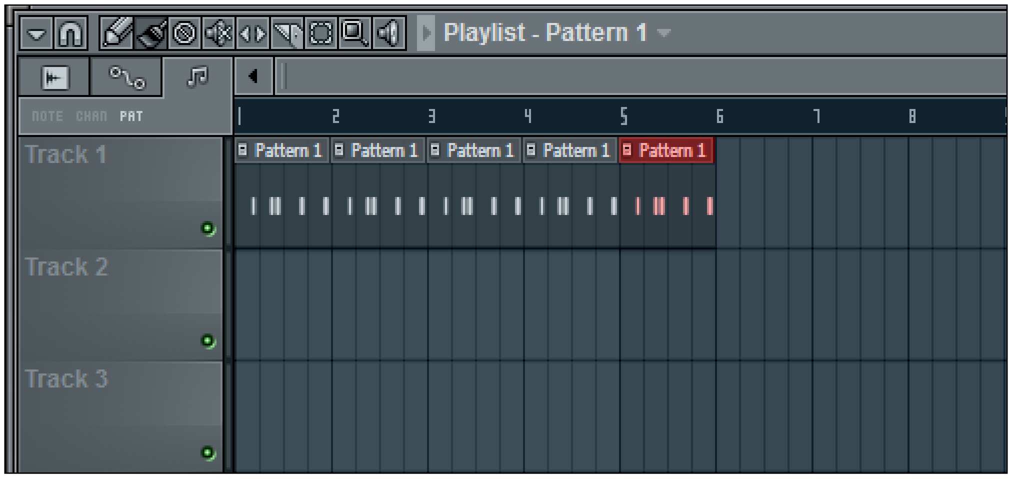
- **Kick** can be compared to your big drum.
- **Clap** is an approximation of the clapsticks. Clap (also called tala) itself is a term used in Indian classical music for the rhythmic pattern of any composition.
- **Snare** represents a smaller drum.
- **Hat** is the cymbal of your drumline.

The following screenshot shows the **Pattern 1** sequence:



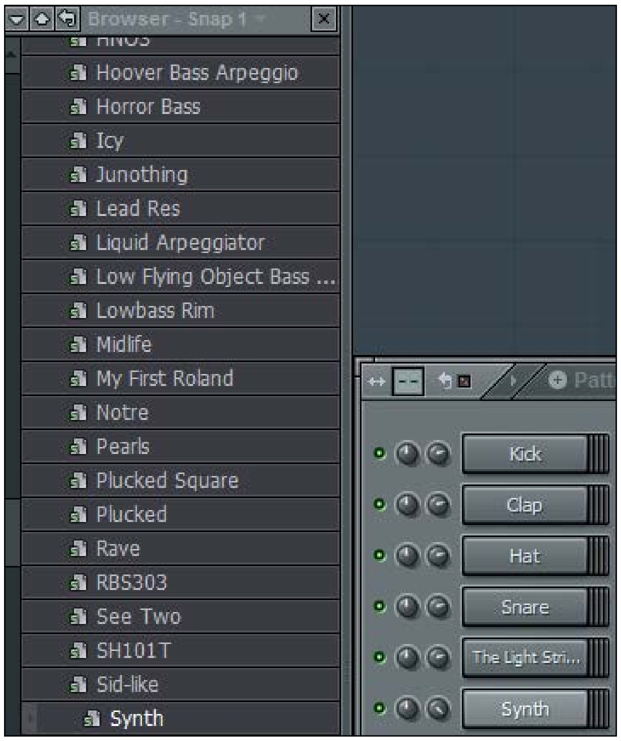
A sequence of rectangles is present for every instrument in your pattern. By clicking on a rectangle you tell FL Studio to activate this instrument at that particular point. Right-clicking on a highlighted rectangle will turn it off again. Almost everything in FruityLoop studio is turned on or added by left-clicking, while right-clicking is used for turning off or deleting. Try making a nice-sounding drumline by activating some of the instruments at particular time intervals.

Once you have created a pattern, it can be added to a playlist. The **Playlist** console will hold all the pieces of music for your project as shown in the following screenshot:



All your patterns can be cued or played simultaneously or sequentially, depending on how you use the different tracks it provides. When left-clicking on a spot in the **Playlist** console you basically *paint* a pattern on that spot. Right-clicking on a pattern will remove it. To change the pattern, you are currently placing a drop-down box at the top-right corner of the **Playlist** console.

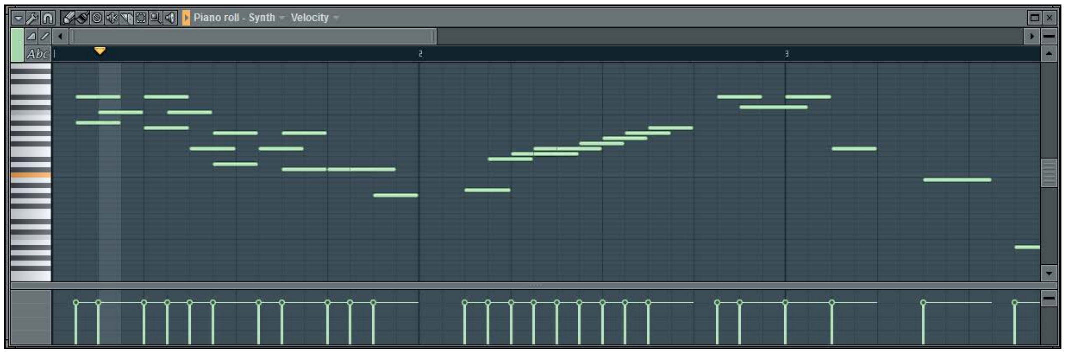
FL Studio provides the user with a wide array of instruments, sound effects, and even premade music and some voice effects as shown in the following screenshot:



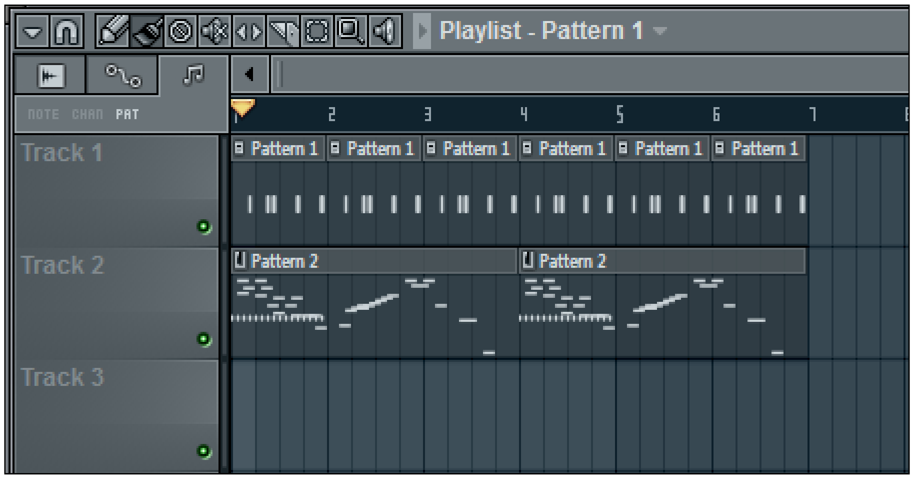
All these resources can be accessed with the **file browser**. From here you can add an instrument to your sequence builder, for instance, a synthesizer or a guitar. Every sound type has a different symbol as shown in the following screenshot, and the premade music can even be previewed (or heard in advance) in the browser itself:

Adding precomposed melodies allows you to quickly make a rather decent song, which in turn can be incorporated into your game.

If you have added an instrument, such as the synthesizer to your sequencer, try opening the **Piano roll** console for it as shown in the following screenshot:

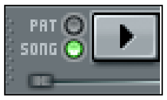


The **Piano roll** console allows you to define every note that the instrument needs to play. For some instruments, such as a drum, this is not always necessary but for others, it's definitely recommended to make your own little tune in the **Piano roll** console. You can do this in the same pattern as your drumline, or you could start a different pattern and unleash your creativity there as shown in the following screenshot:

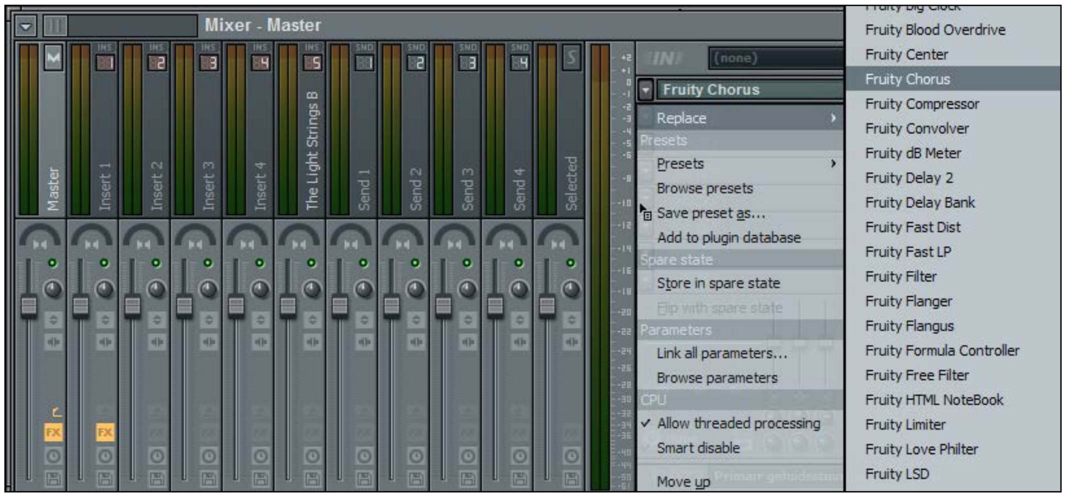


Eventually, every piece of music you create should end up in the playlist. Working with different tracks is the key to keep a good view on all the things that are going on at the same time. If you forgot to allocate your different instrument categories to different tracks, don't worry, there is an option for splitting them in the **Playlist** window.

At some point you will want to hear how your different tracks sound when played together. For this, you need to switch from the pattern to the **SONG** mode as shown in the following screenshot:



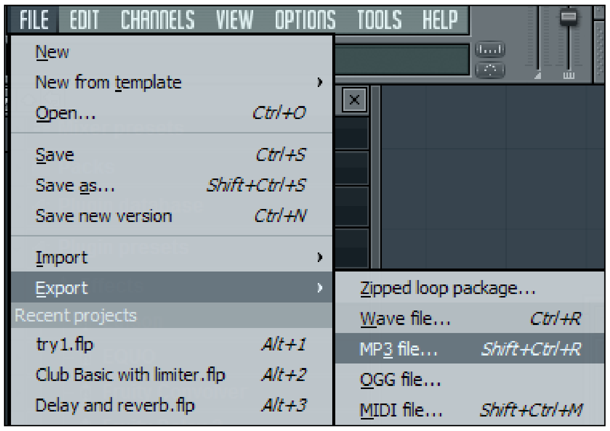
If you feel that some extra tweaking needs to be done to your different instruments, this is where the **Mixer** console comes into play. The **Mixer** console allows you to change things such as volume, balance, and special effects as shown in the following screenshot:



Adding a special effect or filter to the music can quickly give you that extra touch you are looking for. There are many preset filters to choose from and they can all be tweaked separately. If you are looking for a quick fix, you are of course free to leave them at their default settings and work with that.

In every single one of these four elements: sequencer, playlist, piano roll, and mixer, some templates and/or defaults are available. Always look for these in case you don't want to put too much effort into creating your own music. You can use what is already there, tweak it a little, and you have your own soundtrack in no time!

When you are done with your first song you might not only want to save it, but also export it as both `.mp3` and `.ogg` files.



Again, don't forget to put the project to the song mode instead of the pattern mode, or you will only export the pattern you had currently selected.

Once the song is exported you can use what you just created in ImpactJS.

Adding background music to your game

Background music is something that you will want to be played at all times. A lot of games change the music from a calm to a more upbeat track when the going gets tough. All these things can be done using the `if` conditions either in your main code or in a separate file dedicated to managing your playlists.

ImpactJS has two important classes that are responsible for all the sound you will want to use: `ig.music` and `ig.sound`. `ig.music` is what we need for our background music. Let's say you want to add your music to the projects of *Chapter 3, Let's Build a Role Playing Game* or *Chapter 4, Let's Build a Side Scroller Game*. Add the following code to `main.js` in the `MyGame` definition's `init()` function:

```
init: function() {
  this.loadLevel(LevelLevel1);
  ig.input.bind(ig.KEY.UP_ARROW, 'up');
  ig.input.bind(ig.KEY.DOWN_ARROW,'down');
  ig.input.bind(ig.KEY.LEFT_ARROW,'left');
  ig.input.bind(ig.KEY.RIGHT_ARROW,'right');
  ig.input.bind(ig.KEY.MOUSE1,'attack');
  var music = ig.music;
  music.add("media/music/background.*");
```

```
    music.volume = 1.0;
    music.play();
  },
```

Notice that we added our song as `background.*`, not as `background.ogg` or `background.mp3`. This way the game knows that it needs to look for all files called `background`, regardless of their extension. Since we made a separate `music` folder within the `media` folder, we should have no naming conflicts here. Using `background.*` is not only conveniently short (one line of code instead of two), but also helpful to the system for using the `music` file which it is able to play. Sometimes this will be `.mp3`, and sometimes `.ogg`; at least now the `music` file to use can be determined automatically. Chrome now seems to prefer WebM over `.mp3` or `.ogg` but will still work with `.mp3` and `.ogg`. Firefox, on the other hand, prefers to use `.ogg` and doesn't work with `.mp3`.

`ig.music` is in itself a sort of playlist with several functions. Using the `add()` method will add another song to the end of the playlist. You can populate this list with virtually an endless number of songs. The `music.volume` method sets your song volume and ranges from `0` to `1`. The `music.play()` method will activate the first song from the playlist. The previous piece of code will not only activate your song, but also loop it indefinitely because it does this by default. There are many other functions apart from the ones used to simply start the loop.

`fadeout(time)` will make your song fade out over the time you specified as input argument. When the song's volume reaches `0`, it will call the `stop()` method, which stops the song from playing. Whatever you expect on a regular radio is present in ImpactJS. You have the `pause()` and `next()` methods, and the `loop` and `random` properties to make the songs loop and play randomly. Another interesting property is `currentIndex` since this will return the position of the currently playing song in the playlist. This is extremely useful in managing the order of your songs and switching between them when necessary.

Playing a sound when an action has happened

`ig.music` is convenient for music use since it has many functions in common with a basic media player. While for playing music `ig.music` is the best option, for playing sound effects, you should use `ig.sound` for best results.

A sound effect is not constantly active but only happens when certain actions are performed. Let's say we want to hear a gunshot when the player fires a projectile. We need to add the sound in the player's `init()` method so it becomes available as a resource.

In `player.js` add `this.gunshotsound` using the following code:

```
init: function( x, y, settings ) {
  this.parent( x, y, settings );
  // Add the animations
  this.addAnim( 'idle', 1, [0] );
  this.addAnim('down',0.1,[0,1,2,3,2,1,0]);
  this.addAnim('left',0.1,[4,5,6,7,6,5,4]);
  this.addAnim('right',0.1,[8,9,10,11,10,9,8]);
  this.addAnim('up',0.1,[12,13,14,15,14,13,12]);
  //set up the sound
  this.gunshotsound = new ig.Sound('media/sounds/gunshot.*');
  this.gunshotsound.volume = 1;
},
```

We then actually play the sound when the projectile is fired by adding the following code to `player.js`:

```
if(ig.input.pressed('attack')) {
  if (GameInfo.projectiles> 0){
    ig.game.spawnEntity('EntityProjectile',this.pos.x,this.
pos.y,{direction:this.lastpressed});
    ig.game.substractProjectile();
    this.gunshotsound.play();
  }
}
```

While in `ig.music` songs are added to a playlist, sounds are initiated by calling a new instance of `ig.sound`. When only one song is added to the music playlist, by default it is looped forever. This is not the case for the sound effects initiated with `ig.sound`, there is no `loop` property, and sounds are thus played only once when the `.play()` method is called. `ig.sound` has the property `.enabled`, which is set to `true` by default. Setting it to `false` will deactivate all sound and music for the game. This is useful since some mobile devices still have trouble when they need to play two different sounds at once. Playing two different sounds at the same time is extremely common, especially if you already have your background music playing at all times. By using Ejecta, the ImpactJS direct canvas solution, this issue can be resolved. The code remains the same but Ejecta currently only supports iPhone and iPad, not Android or Windows devices.

Tips when using sound files in your game

Optimizing sound files is about keeping it short and simple. Most games have short songs which aren't too intrusive so they don't always get noticed. A song that doesn't get noticed can still affect a mood while it doesn't get too repetitive. For optimization purposes a few are a few things you should certainly take care of:

- Keep the songs short and load them into memory only when needed.
- Prepare the same song in both `.ogg` and `.mp3` so that the system on which it needs to be played can choose which extension is most efficient.
- Double-check whether your sound works on the mobile devices on which you aim to release the game using the following code. If it doesn't, make sure to turn off all the sound on these devices until it becomes possible to make the sound work on them.

  ```
  if(ig.ua.mobile){
    ig.music.add("media/music/backgroundMusic.ogg");
    ig.music.play();
  }
  ```

- This is not so much an optimization as a user-friendliness measure, but make sure you allow the player to turn off the music and sound effects. Best is to separate both: some players like to hear their guns fire but just don't like your music. If you use game analytics, make sure to track these changes so that you can learn which kind of songs are acceptable and which ones aren't.

Summary

In this chapter we have discussed music and sound effects as an important element that aids in creating an atmosphere in your game. We talked about whether you should buy or create music and where you can find it: for free or at a price. We created our own basic background tune using FL Studio and added this to our game. Finally we went over some pointers for using music in ImpactJS.

In the next chapter we will take a look at graphics. We will check if it is better to buy or to make them, and how to create graphics with Inkscape or Photoshop.

7
Graphics

You can have the perfectly working script, but there is no game when there is nothing to see. Graphics are important and here we will look into how to get them and implement them in ImpactJS. In this chapter you will learn:

- The types of graphics present
- What you should consider while deciding whether to make or buy them
- How to make vector graphics using the free tool, Inkscape
- How to turn reality into a game with the use of Adobe Photoshop

Game graphics have been evolving since the start of digital gaming. Take a quick peek at Spacewar! and its arcade version, Computer Space, Pong, Gun fight, and many other games from ancient times. The first thing you will notice is not the difference in gameplay but the lack of graphical splendor. The development of faster computers and dedicated graphical processors has made games increasingly pretty to look at. There was certainly a general trend towards more realism: how much can we make a game look like a real-life one without burning our processors to a crisp? It's a bit like what happened with paintings in general. Painters tended to strive towards more detail, a better approximation of what you would see in real life. It was a challenge to do, until they started using optical lenses to reflect an image directly to the canvas. All they then had to do was trace and color it. Artists started looking at new ways to express themselves on a canvas since perfection was not a guarantee for success any longer. Centuries after graphical perfection was reached, the world saw paintings like Pablo Picasso's Guernica and the Scream of Edvard Munch. Neither of these is even close to reaching perfect realism; they both have something to intrigue people though.

Something similar seems to be happening in the gaming world. Recent games have proven we can come terrifyingly close to realism, and some game developers have begun looking for more original looks. For example, Nintendo has never strived to come close to delivering realistic graphics, yet their skill in producing great games is revered all over the world. This is because they understand that stirring up a certain feeling in the player is more important than showing the same thing a player would see by looking away from their screen.

Take a look at Yoshi's Island, a game released for Super Nintendo in 1995. The depicted scenery here is far from realistic. Yet, just playing it for 10 minutes fills you with a feeling of joy. Everything looks so happy and sparkling, with bright and happy colors. When they don't intend to kill you, animals and even clouds smile at you with a sincere cheerfulness.

Zelda: The Wind Waker, released in 2003, was one of the first big games using cell-shaded graphics. Cell-shaded or toon-shaded graphics appear as if they are hand drawn. Cell shading has been used by quite a number of other very successful games like Borderlands and Okami.

The previous examples were 3D games, but the very fact you are now reading this is proof that making a game is not about graphics alone. Many years ago, games made the transition from 2D to 3D with great success. Even our beloved flat-lander Mario was able to make its transition splendidly. 3D games are generally regarded as being more pleasant to look at than 2D games. Yet, here you are, preparing yourself to make a 2D game. It is proof that nice graphics are important to convey a certain feeling, but conveying this is possible in any form you wish, like art itself.

Making/buying the graphics

Do we need to buy or make our own graphics while making a game? We are lucky to at least have a choice in this. For 3D games the option for making graphics yourself is often restricted by the size of your development team. With 2D games, the option of doing it all on your own remains a realistic one. If you have no budget for buying sprites and tilesets, you have three major options for creating your graphics:

- Pixel art
- Vector art
- Creating reality using Photoshop

Of these three options, drawing your characters and scenery yourself pixel by pixel is the most ambitious one. Great artists can get really nice results with this, but even the most experienced pixel artist will spend several hours on just a few characters and tilesets. There are several tools out there to help you transfer your own drawing skill to the PC, such as the digital drawing pen and software: Adobe Photoshop or it's free counterpart GIMP. If you have no experience in drawing in general and don't feel a strong urge to commit yourself to this in any way, then simply don't.

The second option is vector graphic design. Vector graphics differ from pixel art in that drawings are built up with lines and basic forms, not individual dots. These lines and forms are freely scalable to both higher and lower resolutions, whereas for pixel art, this can be very difficult. Building up drawings from basic forms such as the rectangle, circle, and lines require a different kind of insight than regular drawing does. The prerequisite to making graphics is basically transferred from needing a steady hand to having an analytical view on objects and life forms. Take the birds from Angry Birds as an example. Their basic form is a circle with circular eyes placed in the center. Their beaks are somewhat rounded triangles and their eyebrows and tails are nothing more than a collection of rectangles. If you look at these angry birds from this more analytical point of view, it becomes easier to draw one yourself. If you feel you have that somewhat analytical insight, even if your drawing skills are only average, you can make your own tilesets if you put in enough effort. In this chapter there will be a short introduction on how to do this.

The last option is more of a quick fix. By taking pictures of objects and turning those into tilesets, you can quickly have some graphics at your disposal. While getting close to a realistic scene is very difficult for 3D games, it's actually the easiest way to go in 2D games. The main problem here of course is the fact that you cannot distinguish yourself easily from competitors if you use adjusted pictures, which is a real drawback while marketing your game. Nevertheless, the graphics look nice and it's a quick and cheap way to get them.

Where to buy graphics

Even though 2D games are quite common, there are not many companies out there that are specialized in supplying tilesets to hobby game developers. Most game artists either work for the game company itself or work on a custom basis, which often becomes too expensive for someone developing games in his or her free time.

However, there are a few affordable 2D game graphic producers out there, for example, `www.sprites4games.com`. They have some free sprites available but they are especially praised for their beautiful yet affordable custom work

While downloading free tilesets from random web pages, there are two main issues:

- The tilesets are very incomplete, so they don't actually allow you to create an entire game out of them.
- The other problem with free tilesets is that they are not actually free. They are often ripped from an existing game and reusing them is illegal.

For example, on `www.spritedatabase.net`, you can download the tilesets of entire games. But using them for actually publishing your own game will probably result in getting sued for copyright violation.

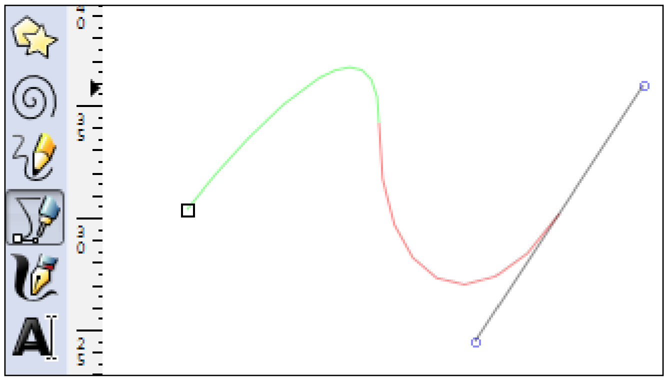
Sometimes you can also find tilesets on the bigger art and photo websites, such as `www.shutterstock.com`. The problem here is clutter; it's difficult to find actual game graphics between all those other pictures. If you finally found some, you face the same problem as with the free ones: incompleteness. You can, at that point, contact the artist and request some more graphics but then it becomes custom work again, which can often become rather expensive.

Introduction to vector graphics

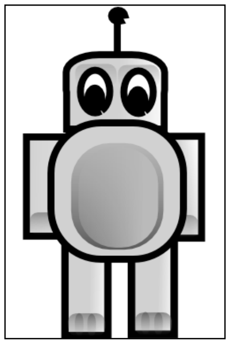
Now that we have taken a look at our different options, let's dive a little deeper into one of them: creating our own vector graphics. There are several interesting tools out there to help you with this. Adobe Illustrator is one of the best on the market. However, here we will use a somewhat less advanced but free tool: Inkscape. You can download Inkscape on their website: `www.inkscape.org/download/`.

Once we have installed Inkscape on our computer, we can go ahead and make ourselves a robot character.

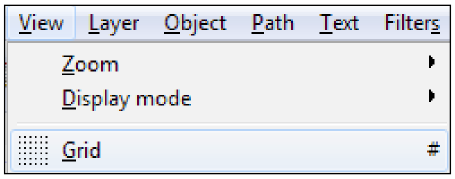
There are several ways to draw yourself a character or an object. The true artists do so using the pen tool as shown in the following screenshot:



This is a very versatile drawing tool, which enables you to draw both straight lines and the most perfect symmetrical curves. However, in this short beginner tutorial, we will limit ourselves to using basic forms such as rectangles and circles to construct our small robot, as shown in the following diagram:

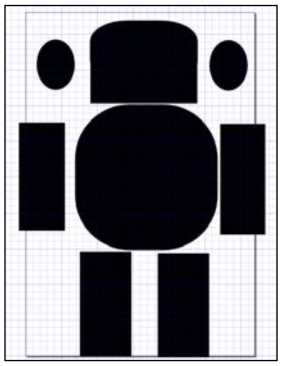


It is effectively going to be a small robot since we will want it to be of the same size as the characters we have been using: 48 x 32 pixels. Even though vector graphics are scalable, it is still best to work on the scale you are going to use them in. While working with these small resolutions, it also makes sense to actually see which pixels you are going to fill. You can do this by turning on the **Grid** option under the **View** tab. Also, you will need to switch between a zoomed-in picture and its actual size; this way you will keep a sight on how much detail you are actually going to put in the game. Zooming in and out can be done using the *Ctrl* key and the scroll wheel of the mouse; also, the shortcut to see everything on a 1:1 scale is simply achieved by pressing the *1* key on your keyboard.

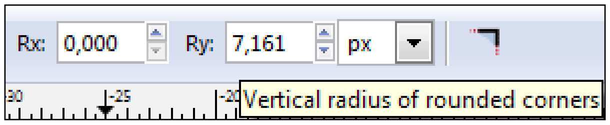


When we take a look at the robot we want to build, something important can be noticed: the head is blown out of proportion. Normally, a human head would be between one-eighth or one-seventh the size of the human body. When drawing for a low resolution, it is a good idea for the head to be about one-third to one half the size of the body. It's very unrealistic but at least you will be able to see some facial features, such as the eyes and mouth. This big-headed style is called Chibi, which means a *short person* in Japanese; it's ideal for small animations.

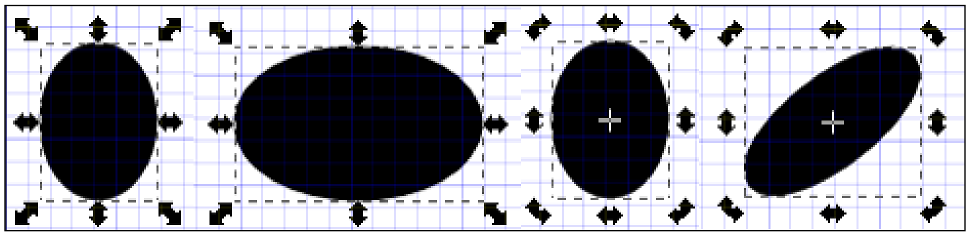
Let's first have a look at the basic forms that we will need. This seems to be not much more than a few rectangles (both rounded and regular ones) and two ellipses for the eyes, as shown in the following screenshot:



A rectangle's corners can be easily rounded by selecting the normal rectangle and changing the radius of its corners with the help of the panel shown in the following screenshot:

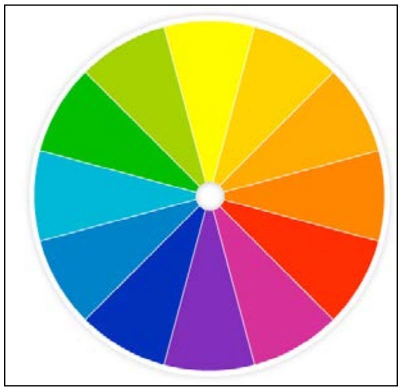


An ellipse is nothing more than a stretched out circle. You can stretch any form in any direction and rotate or skew it if necessary, as shown in the following screenshot:

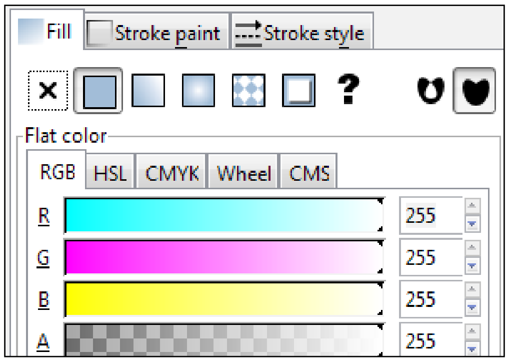


While working with vector graphics, it is best to have different layers in order to cope with different animations. For example, if we want our robot to walk, we will need its one arm and leg to stick out, and then its other arm and leg. Putting the body and the arms and legs in separate layers makes sense from an animation point of view. The body's formation does not change when moving, while the limb's formation does.

Now that we have our basic forms, let's focus on colors. While working on low resolution, it's best to get a big contrast going. You can do this by either having a color close to white and one close to black, thereby playing with brightness. Alternatively, you can choose to work with two complimentary colors. Two colors are compliments when they are each other's opposite, resulting in the biggest contrast when put next to each other. So when picking colors, it's useful to bring in the color wheel. Colors on opposite sides of each other in this wheel are considered complimentary colors. For example, the compliment of yellow is purple, as shown in the following diagram:

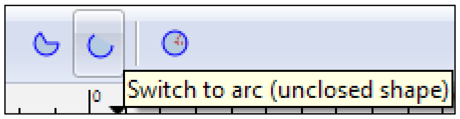


Our robot is going to be gray and black. In order to color it we just need to right-click our mouse button on each of the elements, select **Fill and stroke**, and fill it with the color we like.

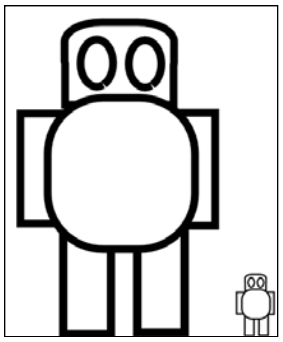


R 255
G 255
B 255
A 255

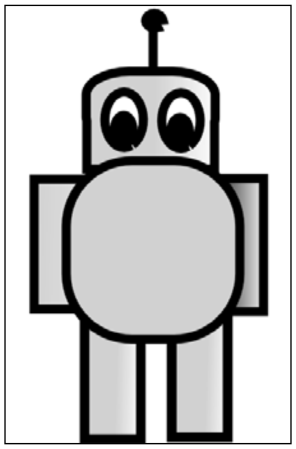
In addition, we can give our robot a small extra detail in his eyes by switching our circles to an incomplete arc, using the panel shown in the following screenshot:



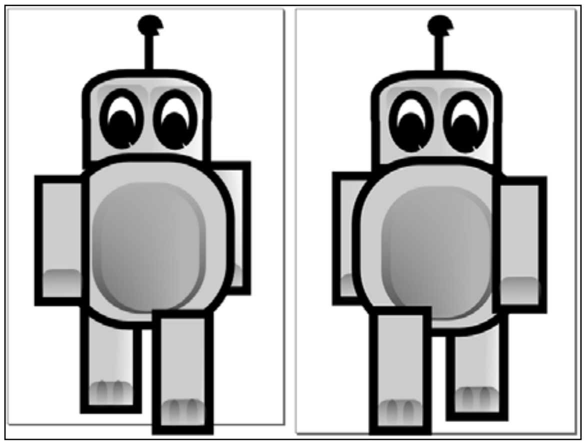
Our robot now has a recognizable form and even has this small eye detail. The problem with these details is that they will not always be visible when actually playing your game, as shown in the minimized form of our robot character in the following diagram; finding the right amount of detail can be tricky.



We can add the antenna on his head as shown in the following diagram, and although it's small, it will still be recognizable; ultimately this is what you need to take into account for every detail you add. Let's introduce a little bit of shadow to the drawing shown in the following diagram of the character. We can do this by changing the fill to a gradient pattern instead of an even fill.



In addition, by adding some extra forms with these gradient shadow patterns, we can make the design look even more realistic. As an exercise, you could add your own animation for when the character is idling. For instance, a person would inhale and exhale, making his or her chest go up and down. To portray this, you can add one extra image, making the game feel even more alive. Eventually, we end up with our final robot. Teaching it how to walk is then a matter of putting the best foot forward, followed by the other one of course. If you work in a single layer, it can be done by bringing a leg and arm to the front by selecting them and pressing the *Home* key. Pressing the *End* key will position the selected arm behind the other forms. Alternatively, you can use the **Object** menu to achieve the same thing. Ideally though, you would want to work with different layers, since this makes life a lot easier. However, we won't go into that level of detail here.

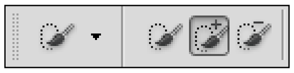


The robot looks as if he is going to leave his little picture and come right towards you, as shown in the previous diagram. To get a full character you will need to do the same for at least one profile view and its back. This can be done quickly once you are experienced in doing so. However, there is a quicker way to get graphics. Though, you will probably only want to use them as placeholders until a drawing is ready, it's still a nice option. This option is using Adobe Photoshop for real life pictures.

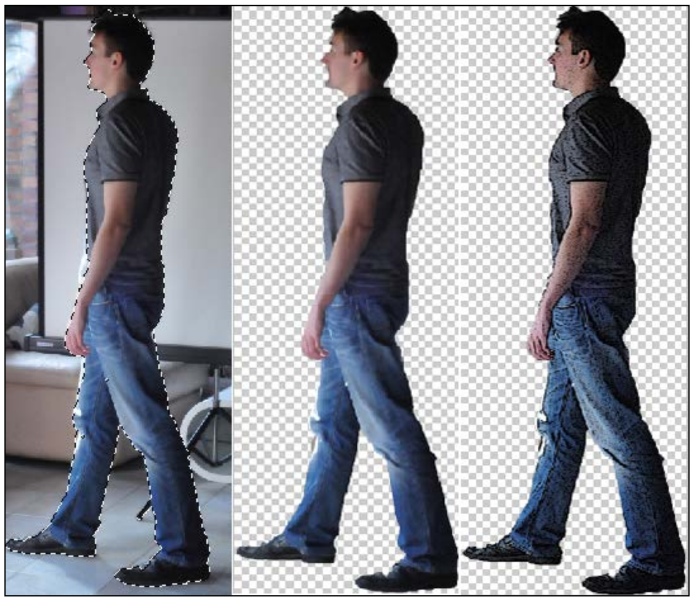
Creating your own avatar using Adobe Photoshop

Ever dreamt of having yourself walk about in your own game? Well, now you can! All you need is a camera and a tool similar to Adobe Photoshop. While we will use Adobe Photoshop, there are plenty of free replacements out there that will do the trick just fine. Even the browser solutions are quite decent. Pixlr is a good example of this. It can be found at `www.pixlr.com`.

We will have to start off with a bunch of pictures taken from all relevant directions. It will be best to take them in front of an evenly colored screen; a simple white blanket or wall will be fine too. If your background is easy to distinguish from the person you would like to capture, subtracting him or her from the picture will be easier. We can do this with the quick selection tool as shown in the following screenshot:

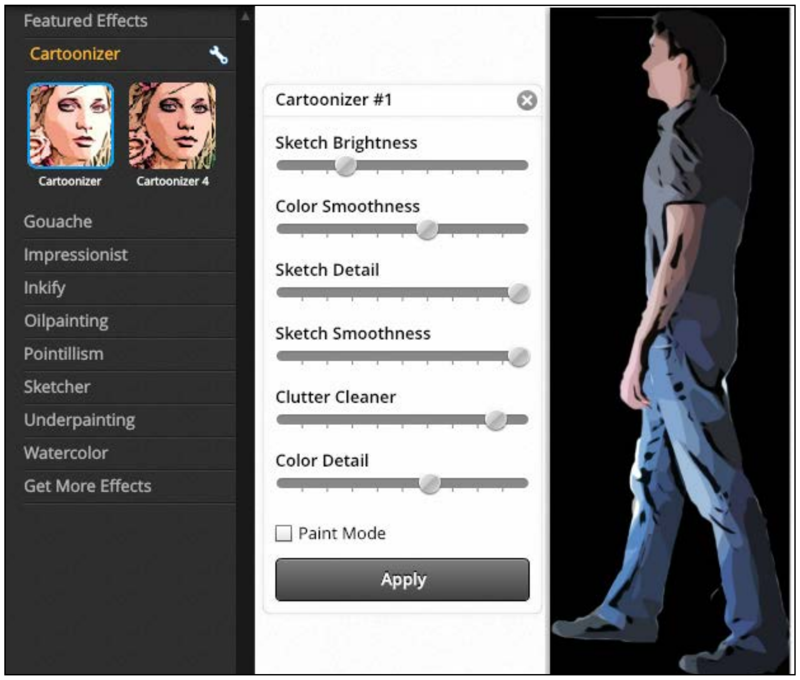


After separating the person from the background, we can simply put the picture in a new file with a transparent background and even add some effects to it in order to give it a more surreal touch, as shown in the following screenshot:



Don't restrict yourself to what Adobe Photoshop has to offer. There are good websites out there that can transform your picture in ways you can barely fathom. One such websites is `www.befunky.com`.

Here we have the option to unleash the cartoonizer effect on our picture, rendering the person virtually unrecognizable while it produces this nice cell-shaded style, as shown in the following screenshot:

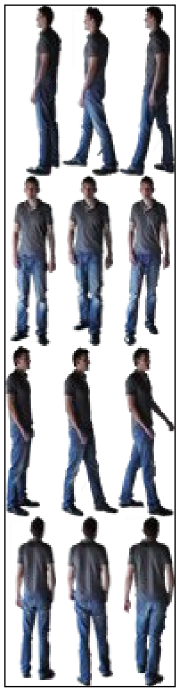


You will have to repeat this process for all your pictures, which can be quite time consuming. However, it will be done much more quickly than actually drawing them yourself. Also remember, passive objects only need one picture. The game characters, needing actual animation sheets, represent most of the workload.

Now that we have our individual sprites, let's take a look at the animation sheet itself. If you don't have the necessary suitable pictures of yourself, now is the time to go and let someone take a few pictures of you posing in front of a white wall. There is something peculiarly odd about seeing yourself inside a video game, so try it out.

Adding your creation to the RPG

In order to get from your individual sprites to a fully fledged animation sheet, all that needs to be done is to order them nicely in a single file.



Before getting into this, you will need to think about how big you need your images to be. In this example they are 32 x 96 pixels in size. In the earlier chapters, we had characters that were 32 x 48 pixels in size. Having drawings that are less stretched than our current example is preferable, since they will allow easier gameplay. It is much easier to maneuver a character with the dimensions of a circle or square than one that is long and thin. However, in this case our test person is long and skinny and we would like to keep him that way.

Actually arranging the pictures on a single animation sheet is a work of precision, so it is recommended to work with the picture's coordinates. Adobe Fireworks allows very intuitive handling when it comes to arranging drawings by setting their coordinates. Any picture program should do the trick though; you could even do this in MS Paint. There are other options out there of course. A spritesheet generator would make arranging sprites and saving them as a tileset considerably easier. You also have the option of using some JavaScript arranging code in Fireworks to automate the positioning process. However, there will be no elaboration on these topics here.

When you are finally done setting up your own sheet, it is time to introduce it into your game. Save the file as `player.png` and replace the code and sheet in one of the previous chapters of your choice.

```
animSheet: new ig.AnimationSheet( 'media/player.png', 32, 96 ),
init: function( x, y, settings ) {
  this.parent( x, y, settings );
  // Add the animations
  this.addAnim( 'idle', 1, [4] );
  this.addAnim('down',0.1,[3,4,3,5]);
  this.addAnim('left',0.1,[0,1,0,2]);
  this.addAnim('right',0.1,[6,7,6,8]);
  this.addAnim('up',0.1,[9,10,9,11]);
}
```

Our animation sequence is pretty short. For each of the perspectives, we switch between standing still and moving either the right or left leg forward. The sheet could have been even smaller if we had a perfectly symmetrical character. We would then only need the animation for either walking to the left or the right and get the other animation by flipping the image as seen in an earlier chapter.

Hints for graphics in HTML5

To end this chapter, let's go over a few pointers for using graphics in HTML5:

- Try to keep your animation sheets as small as possible. There is no need to duplicate certain sprites; the animation sequence allows the same sprite to be addressed multiple times if necessary. It's also good to know there are image size limits that differ for every browser, though you would have to be rather sloppy to reach this limit.
- Use a file format that supports transparent backgrounds. A PNG file will do the trick. JPG is not capable of saving a background as transparent and will instead interpret it as being solid white.
- Try to work with symmetrical figures where possible. This way you can flip the image to make a character walk from left to right and vice versa with the same images. It also reduces the number of sprites you need and thus the effort to make them.

- When using background maps in ImpactJS, it can be useful to prerender them. A background map differs from a regular level layer in the way it is drawn by a code array you supply in your scripts instead of the standard JSON-coded level files. This way it is possible to set up recurring backgrounds.

```
var backgroundarray= [
  [1,2,6],
  [0,3,5],
  [2,8,1],
];
var background = new ig.BackgroundMap( 32, backgroundarray,
'media/grass.png' );
```

- Prerendering the background will make the system create chunks, which are a collection of tiles. Choosing to prerender will require more RAM since larger pieces need to be kept in memory but will speed up the drawing process; this way there is less taxation on the device's processor. Knowing that you have this option and depending on whether you think RAM or processing power will be the bottleneck, you can choose to either prerender your background or not by use of the ImpactJS `.prerender` property. In addition you can set the chunk size to fine-tune the balance between the two resources:

```
background.preRender = true;
background.chunksize = 4096;
```

Summary

Graphics are an important element of any game as they are the visualization of everything your game represents. Though there has certainly been a trend towards more realistic graphics in games, it is not absolutely required for a good game experience. We talked about whether you should make or buy your graphics and where you can buy custom graphics at an affordable price. We distinguished three important options if you decide to create your own graphics: pixel graphics, vector graphics, and using Adobe Photoshop reality. Skipping the first option, we took a quick look at how to develop vector graphics with Inkscape and add yourself to your game using Adobe Photoshop. The chapter ended with some hints on working with graphics in the game. In the next chapter, it's finally time to show our game to the world as we get to deploy it to several distribution channels ranging from a regular website to Google Play.

8
Adapting Your HTML5 Game to the Distribution Channels

When your game is finally ready for the entire world to see, it is time to consider the possible distribution channels. Do you want people to play your game in their browser on a website or as a web app? Maybe you want them to play it with a tablet or smartphone, either in a browser or as an app. In this chapter, we will look into several of those different options and what needs to be done for a successful implementation.

In this chapter, you will learn:

- Preparing your game for the web browser
- Making adaptations for mobile web browsers
- Releasing your game as a Google Chrome web app
- Turning the game into an Android app
- Making your game playable on Facebook
- Implementing AppMobi's direct canvas

Preparing your game for the web browser

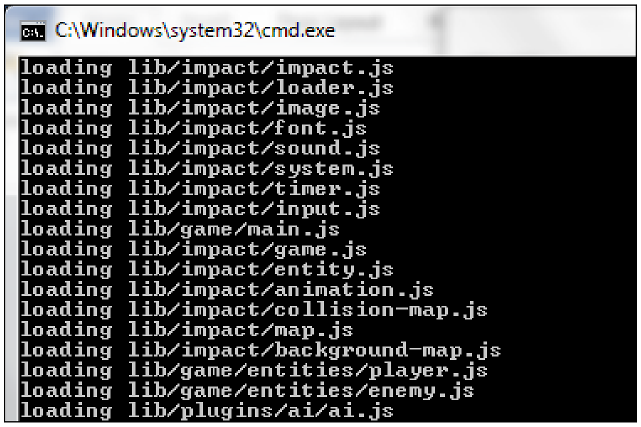
You have been testing your game in a web browser during development. So what is the difference between your local server and a public or production server?

Just before releasing your game to the public, you will need to bake it. **Baking** the game is not more than compressing the code. This has two advantages:

1. Compressed code will load faster into the browser than uncompressed code. A shorter loading time is always a big advantage, especially for people who play your game for the first time. These people don't know yet that your game is extremely awesome and don't want to waste time looking at a loading bar.
2. The baked code is also harder to read. All your different modules, neatly ordered in separate files, are now in a single file together with the ImpactJS engine. This makes it harder for the average Joe to just copy and paste your precious source code from the browser and use it in his or her own games. However, it doesn't protect against people who really know what they are doing; the code does not become encrypted, just compressed.

The tool for baking your game comes with the ImpactJS engine you have downloaded. In the `tools` folder in your game's `root` directory, you should have four files: `bake.bat`, `bake.php`, `bake.sh`, and `jsmin.php`. Follow the following steps to bake your game:

1. Open the `bake.bat` file with a text editor and you will find the following line:
   ```
   php tools/bake.php %IMPACT_LIBRARY% %GAME% %OUTPUT_FILE%
   ```
2. Change `php` to the directory of your `php.exe` file in your XAMPP or WAMP server. For a default XAMPP installation, this line will now look as follows:
   ```
   C:/xampp/php/php.exe tools/bake.php %IMPACT_LIBRARY% %GAME%
   %OUTPUT_FILE%
   ```
3. Save and close the `bake.bat` file and double-click on it to run it. On Windows, a command window will open and a `game.min.js` script will be created in your game's `root` directory, as shown in the following screenshot:


C:\Windows\system32\cmd.exe

```
loading lib/impact/impact.js
loading lib/impact/loader.js
loading lib/impact/image.js
loading lib/impact/font.js
loading lib/impact/sound.js
loading lib/impact/system.js
loading lib/impact/timer.js
loading lib/impact/input.js
loading lib/game/main.js
loading lib/impact/game.js
loading lib/impact/entity.js
loading lib/impact/animation.js
loading lib/impact/collision-map.js
loading lib/impact/map.js
loading lib/impact/background-map.js
loading lib/game/entities/player.js
loading lib/game/entities/enemy.js
loading lib/plugins/ai/ai.js
```

The `game.min.js` script now contains all our code. All we need to do now is change the `index.html` file in our game's `root` directory so it looks for `game.min.js` instead of the `impact.js` and `main.js` scripts.

Open the `index.html` file and find the following code:

```
<script type="text/javascript" src="lib/impact/impact.js"></script>
<script type="text/javascript" src="lib/game/main.js"></script>
```

Replace the previous code by our new compact version of the code, as shown in the following code snippet:

```
<script type="text/javascript" src="game.min.js"></script>
```

You can now strip your game's folder of all code files except for `index.html` and `game.min.js` and upload it to your server. If you bought your own web space, you can use free FTP programs such as **FileZilla** to do this.

Our game is now ready for distribution and by loading it to the web server, you would already have it available for anyone out there. However, we did not yet take into account browsers on mobile devices. Before we look into that, let us quickly recap.

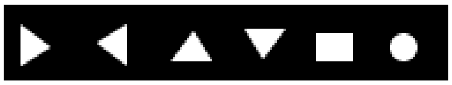
Summing up the preceding content, the conclusions are as follows:

- Before releasing our game to the public, we should bake it. Baking a game is basically compressing the source code. Baking has two big advantages:
    - The game is loaded into the browser more quickly.
    - The code becomes harder to read, thus making it less vulnerable to theft. However, the code is not encrypted making it still pretty easy for a dedicated person to undo the baking.

- In order to bake the game, we change the `bake.bat` file before we run it. This process creates a `game.min.js` script.
- We include `game.min.js` in our `index.html` file instead of `main.js` and `impact.js` before uploading our game to a server.

Preparing our game for mobile web browsers

If you have taken into account the possibility of people using their smartphones to play your game, you've implemented touch screen controls. An example of this can be found in *Chapter 5, Adding Some Advanced Features to Your Game*. However, sometimes this is not enough. You want the player to be able to do the same with his/her smartphone as he would with his/her computer. To make this happen, we can introduce **virtual buttons**. These virtual buttons are areas on the screen which will act as if they are regular keyboard keys. We can create these using **CSS** (**Cascading Style Sheets**) code in the `index.html` file. We can have buttons for every action our player does. In our RPG, he needs to be able to walk in all directions and shoot. In the side scroller game, he can move left or right, fly, and shoot. Let's assume we separate flying from moving upwards. The following screenshot shows our button tilesheet:



Following are the steps to create our virtual buttons:

1. Open the `index.html` file and add the following code below the CSS code for the canvas. If you use the `index.html` file provided with examples of the ImpactJS engine, this file should already include the following style code for the canvas. Also the `index.html` files from both the games in *Chapter 3, Let's Build a Role Playing Game* and *Chapter 4, Let's Build a Side Scroller Game* contain the following CSS code for canvas:

```
.button {
  background-image: url(media/iphone-buttons.png);
  background-repeat: no-repeat;
  width: 192px;
  height: 32px;
  position: absolute;
  bottom: 0px;
}
-webkit-touch-callout: none;
-webkit-user-select: none;
-webkit-tap-highlight-color: rgba(0,0,0,0);
-webkit-text-size-adjust: none;#buttonLeft {
  position: absolute;
  top: 50%;
  left: 10%;
  width: 32px;
  background-position: -32px;
  height: 32px;
  visibility:hidden;
}
```

2. First we define our full button pallet. It has a height of 32 pixels and a width of 192 pixels (six buttons, each 32 pixels wide).
3. Within this button, we can define the six different parts separately. Here you can see the CSS code for the left button. The other five buttons use the exact same code except for their background position since this is their location on the `iphone-buttons.png` image. So, for instance, for the left button the location is `-32`, for the right button this would be `0`, and for the up button it is `-64` as it is third in row. The `webkit` commands are there to keep the layout neat, as intended. If these commands are not supplied, the user could unintentionally change the zoom or colors by tapping the screen.
4. However, we only want our buttons to show on mobile devices. So let's control that with a short piece of JavaScript code in our `index.html` file, as shown in the following code snippet:

```
<script type="text/javascript">
  <!--//test if it is a mobile device-->
varisMobile= {
    Android: function() {
      return navigator.userAgent.match(/Android/i) ? true : false;
    },
    BlackBerry: function() {
      return navigator.userAgent.match(/BlackBerry/i) ? true :
false;
    },
    iOS: function() {
      return navigator.userAgent.match(/iPhone|iPad|iPod/i) ? true
: false;
    },
    Windows: function() {
      return navigator.userAgent.match(/IEMobile/i) ? true :
false;
    },
    any: function() {
      return (isMobile.Android() || isMobile.BlackBerry() ||
isMobile.iOS() || isMobile.Windows());
    }
  };
  function mobileButtons(){
    <!-- show the mobile buttons -->
    if(isMobile.any()){
      document.getElementById('buttonLeft').style.visibility =
'visible';
      document.getElementById('buttonRight').style.visibility =
'visible';
      document.getElementById('buttonUp').style.visibility =
'visible';
      document.getElementById('buttonDown').style.visibility =
'visible';
      document.getElementById('buttonShoot').style.visibility =
'visible';
      document.getElementById('buttonJump').style.visibility =
'visible';
    }
  };
</script>
```

5. In the first part of this script, we define the local variable `isMobile`. It is set to `true` if a mobile device is detected, if not it is set to `false`. In the second part, we set the visibility of our CSS objects to `visible` if `isMobile` is `true`. Remember that their visibility was set to `hidden` when we created them in the CSS part of `index.html`.
6. All that remains to do within our `index.html` file is adding these buttons as the `<div>` elements next to our `canvas` element, as shown in the following code:

```
<body onLoad='mobileButtons()'>
  <div id="game">
    <canvas id="canvas"></canvas>
    <div class="button" id="buttonLeft"></div>
    <div class="button" id="buttonRight"></div>
    <div class="button" id="buttonUp"></div>
    <div class="button" id="buttonDown"></div>
    <div class="button" id="buttonShoot"></div>
    <div class="button" id="buttonJump"></div>
  </div>
</body>
```

The `index.html` file now has buttons which will only show when a mobile device is detected, but it doesn't make our game ready for this just yet. For this, we need to adapt our `main.js` script.

7. Open `main.js` and add the following code to your `init()` method of your `game` instance:

```
if(ig.ua.mobile){
  // controls are different on a mobile device
  ig.input.bindTouch( '#buttonLeft', 'Left' );
  ig.input.bindTouch( '#buttonRight', 'Right' );
  ig.input.bindTouch( '#buttonUp', 'Up' );
  ig.input.bindTouch( '#buttonDown', 'Down' );
  ig.input.bindTouch( '#buttonJump', 'changeWeapon' );
  ig.input.bindTouch( '#buttonShoot', 'attack' );
  //alert('control setup');
}else{
  //initiate background music
  var play_music = true;
  var music = ig.music;
  music.add("media/music/backgroundMusic.ogg");
  music.volume = 0.0;
  music.play();
}
```

8. If a mobile device is detected, the virtual buttons are bound to a game input state. So, for instance, the `<div>` element `buttonLeft` will be bound to the input state `Left`.

9. The code within the `else` statement turns on the background music if you would have that. As addressed in *Chapter 6, Music and Sound Effects*, some mobile devices don't allow sounds to overlap. So, for mobiles, it is wise to turn off the background music so it would not overlap with other sound effects. This will probably not remain an issue forever, but for now it is wise to take into account these sound issues.

10. We will also need to adapt our canvas size so that it fits on the screen of the smartphone or iPad. Replace the default canvas call:

    ```
    ig.main('#canvas', OpenScreen, 60, 320, 320,2);
    ```

    Use the following code to replace the default canvas call:

    ```
    if( ig.ua.iPad ) {
      ig.main('#canvas', MyGame, 60, 240, 160, 2);
    }
    else if( ig.ua.mobile ) {
      ig.main('#canvas', MyGame, 60, 160, 160, 2);
    }
    else {
      ig.main( '#canvas', OpenScreen, 60, 320, 320, 2 );
    }
    ```

11. All this does is initialize the game with different canvas sizes so that it fits on smaller screens like those of iPads (or other tablets) and smartphones. In addition, the intro screen is skipped here; this is a choice and you could leave it in a mobile. Also you could adapt the canvas size for more devices. Here it is shown only for iPad and all other mobile devices, but of course more differentiation is possible.

Hurray! Your game is now ready for mobile! Don't forget to bake it before putting it online; mobile Internet is not as fast as regular Wi-Fi so making your file smaller definitely counts here.

Next, we are going to take a look at making a mobile web app for the **Chrome Web Store** but first let's quickly recap how to prepare our game for mobile browsers.

Summing up the preceding content, the conclusions are as follows:

- If we want our players to have a good game experience on mobile devices, we need to adapt our game interface to accommodate this. We do so by adding virtual buttons.
- The visual aspect of the virtual buttons is created using CSS and an image file in the `index.html`. We can make the buttons visible or hidden, depending on whether the game is played on a mobile device or not.
- In our game's `main` script, we need to bind these buttons to game action states in order to get the same functionalities going as with a keyboard.
- Additionally, we can change the game screen resolution and size so it fits better on the device the player uses.

Turning the game into a web app for the Google Chrome Web Store

A web app is an app which runs in the browser itself, not on the operating system of a mobile device. To release a web app on the Google Chrome Web Store, you need a Google Chrome developer account, which comes at a $5 one-time fee. You will need a Google webmaster account to confirm the ownership of a link Google provides you with. Also, not to make things harder than they need to be, it is wise to get a free AppMobi account. You can do this on their website: `http://www.appmobi.com`. AppMobi is a very interesting beginners' tool for three main reasons:

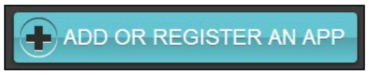
1. They simplify the process to push games to several different distribution channels.
2. They charge nothing for the first 10,000 users of your app or game, allowing you to first make money before they ask you to cut them a slice of the cake; a very attractive pricing scheme indeed.
3. The ImpactJS XDK (cross-environmental development kit) helps adapting the game to different mobile devices by creating artificial viewports. It contains many other useful functionalities such as the simulation location detection.

AppMobi facilitates building a version of your game for the following platforms: iOS, Android, AppUp, Amazon, Nook, Facebook, Chrome, Mozilla, and hostMobi (their own cloud host service).

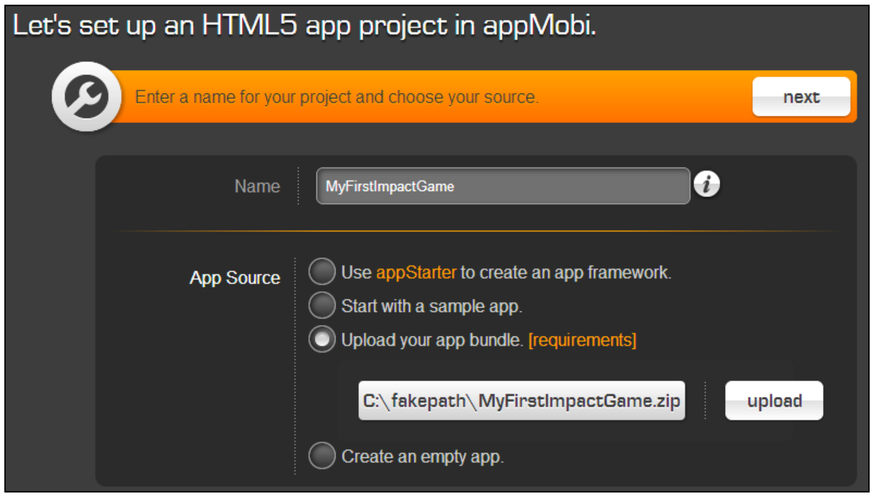
Once subscribed, you can install their ImpactJS XDK for development. On installing the XDK, it becomes easily accessible in your Chrome browser with a plugin icon next to your address bar, as shown in the following screenshot:

You can log in on the appHub: the AppMobi control panel to access all their services. What will be of particular interest to us now is building a Google Chrome game. Following are the steps to build a Chrome game:

1. On first login, you will need to add a new game to your control center by clicking on the button shown in the following screenshot:



2. Give your game a name and upload it in a zipped format to the server, as shown in the following screenshot:



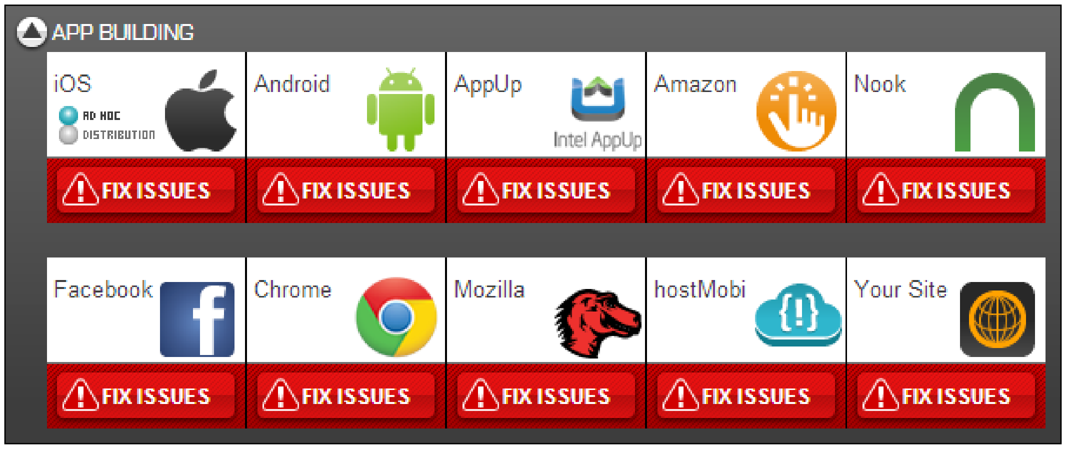
Let's set up an HTML5 app project in appMobi.

Enter a name for your project and choose your source.

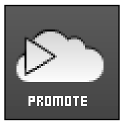
Name: MyFirstImpactGame

App Source:
- Use appStarter to create an app framework.
- Start with a sample app.
- Upload your app bundle. [requirements] — C:\fakepath\MyFirstImpactGame.zip
- Create an empty app.

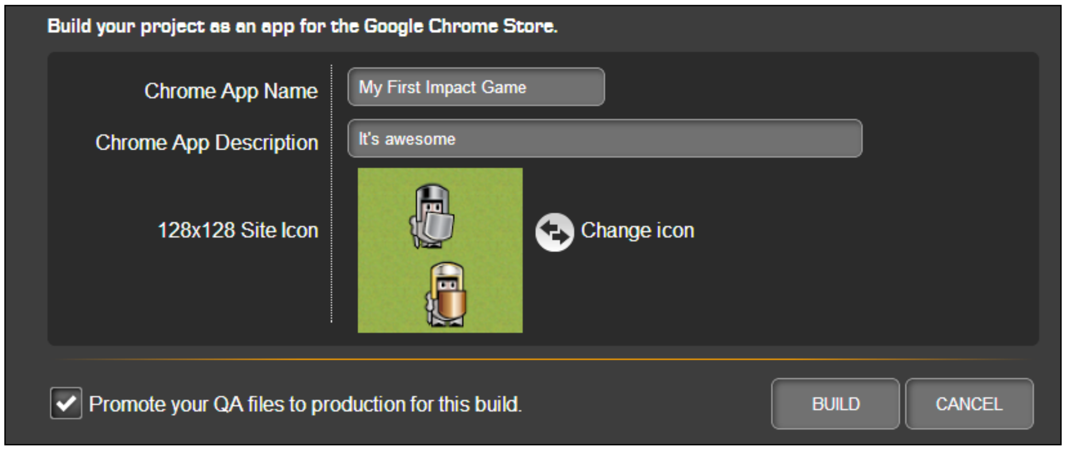
3. You will see the different distribution channels for which AppMobi allows you to prepare a file, as shown in the following screenshot:



4. Before we can build ourselves a Chrome `ready` file, we will need to promote our file to production by pressing the **PROMOTE** button, as shown in the following screenshot:



5. We want to build a game for Chrome, so check out the issues you still have. Chances are you will only need to add an icon for the game. But you do this just before building the game, as shown in the following screenshot:



Build your project as an app for the Google Chrome Store.

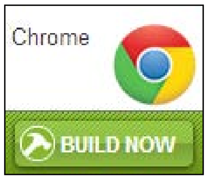
Chrome App Name: My First Impact Game

Chrome App Description: It's awesome

128x128 Site Icon

✓ Promote your QA files to production for this build.

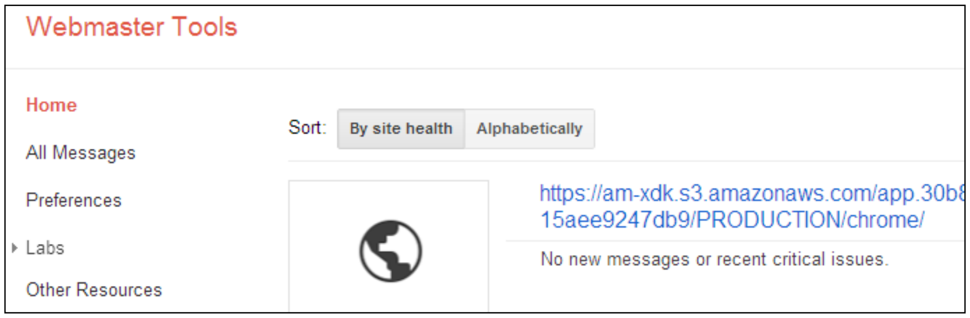
6. If all goes well, you should be able to download a `production` file which you then need to upload to the Chrome Web Store using the button shown in the following screenshot:



7. It is almost time to upload the file to the Chrome Web Store. However, before doing so, open the zipped folder you just downloaded from the AppMobi website and make sure the name of the Chrome icon is exactly the same as is stated in the `manifest.json` file added by AppMobi. This has been a known issue and Chrome will not accept inconsistent naming.

8. If you upload for the first time, you will receive a message saying you need to verify ownership of a domain Google provides you with. To do this, you must insert the tagged HTML file Google allows you to download into the bundle you first uploaded to AppMobi and reupload your game to AppMobi, this time with the verification file inside the zipped bundle. In AppMobi, use the **UPDATE QA** button to upload a new file. Also don't forget to promote to production afterwards.

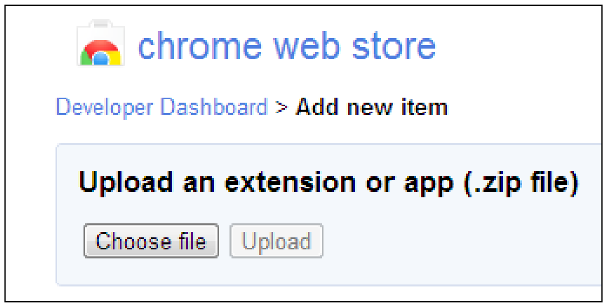
9. In the Google webmaster tool, you add the link provided by Google and verify it, as shown in the following screenshot:



https://am-xdk.s3.amazonaws.com/app.30b8
15aee9247db9/PRODUCTION/chrome/

No new messages or recent critical issues.

10. You can now reupload to the Chrome Web Store and fill out all the necessary elements. You will need to add a detailed description of your game, the pricing scheme, and screenshots using the button shown in the following screenshot:


chrome web store
Developer Dashboard > Add new item
Upload an extension or app (.zip file)
Choose file
Upload

If all goes well, you will be able to beta test your game as a web app as it becomes available for you to add to your Google Chrome. If things go wrong, AppMobi has an abundance of documentation on their services and information on how to use them.

We now have a web app, but we can go through roughly the same process in order to get a real app. In this book, we will take the example of Android. Before doing so, let's quickly recap.

Summing up the preceding content, the conclusions are as follows:

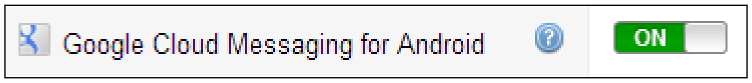
- A web app is an app which is run in the browser instead of directly on the operating system of a device. Google Chrome has such web apps available in its Google Chrome Web Store. Publishing in the store requires a $5 one-time subscription fee.
- AppMobi provides an easy way to build web apps and apps in general. Registration is free, but needs to be paid for once a game reaches a certain amount of success.
- The baked game needs to be zipped and uploaded to the AppMobi server. Here it will be adapted by AppMobi and you can download it again to upload it to the Chrome developer account.
- Google provides you with a link you need to reupload to AppMobi and verify with a Google webmaster account.
- Once the link is verified, you can reupload your game to the Web Store and fill out details such as the game description.
- You can test your game in the browser before submitting it for review and releasing it to the public.

Pushing your game to Android's Google play store

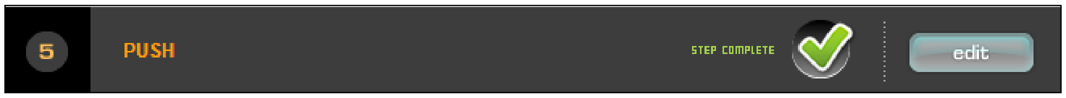
Now that we know how to build a web app, let's build an actual mobile app for the **Google play** store. Once again, we will make use of our AppMobi account to get the job done. However, in addition, you will need a Google developer account, priced at $25 per year, for publishing your game in the Google play store. Following are the steps to push your game to the Google play store:

1. Use the `Upload game` package uploaded to the **AppMobi appHub** or upload a new one.

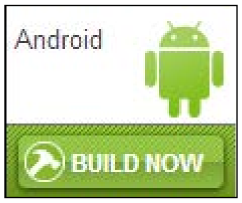
2. Choose **Build** under the **Android** tab and fix any issues you still have. If you succeeded in building the Chrome store web app, there should be only one issue left: setting up Google cloud messaging. For this, you need a **Google Project ID** and an **API key**; you need to get both from your developer account.
3. Sign up for a developer account at `https://play.google.com/apps/publish/signup` or log in if you already have one.
4. Go to your **Google apis** console and create a new project. You will be able to choose your project ID from `https://code.google.com/apis/console/`.
5. Enable **Google Cloud Messaging for Android** in the **Services** section, as shown in the following screenshot:



6. Choose **create a new server key** in the **API Access** section of the control center. After creating a new server, you also receive the API key.
7. Return to the AppMobi appHub where you fill out both the project ID and the API key. You have now set up push messaging. The following screenshot shows the screen after completion of push messaging setup:



8. Your app should now be ready to build. Do so and download the `apk` file by clicking on the **BUILD NOW** button, as shown in the following screenshot:



9. All you need to do now is upload this file to your developer console. Google will require you to fill out a name, description, and add some screenshots. When this is done, you are all set to go.

A word of advice before sending your app for play store review: test on several mobile devices whether the `build` file you downloaded from AppMobi works fluently. This can be done by uploading your file to your own website and downloading it with your smartphone. Alternatively, you can use free cloud storage services such as **Dropbox** to transfer the file from the cloud to any device on which you want to test.

Now we have built both an app and a web app, we will dive into a way to publish your game on **Facebook**. Before doing so, let us quickly recap.

Summing up the preceding content, the conclusions are as follows:

- Building an app with AppMobi is almost the same process as building a web app. However, in order to publish your game as an app, you will need a Google developer account, priced at $25 per year.
- Upload your compressed `game` file to AppMobi appHub if you haven't already. Make sure to receive a project ID and API key from Google apis.
- Build your `android` file and upload it to your developer account where you can send it in for review. However, before doing so, be sure to test your game on a few Android mobile devices.

Making your game available on Facebook

AppMobi could be used to build a Facebook app but Facebook also allows another option of showing your game. You will need a Facebook developers account to go with your Facebook account. There is currently no subscription fee. You can go to the following link to get your Facebook developers account:

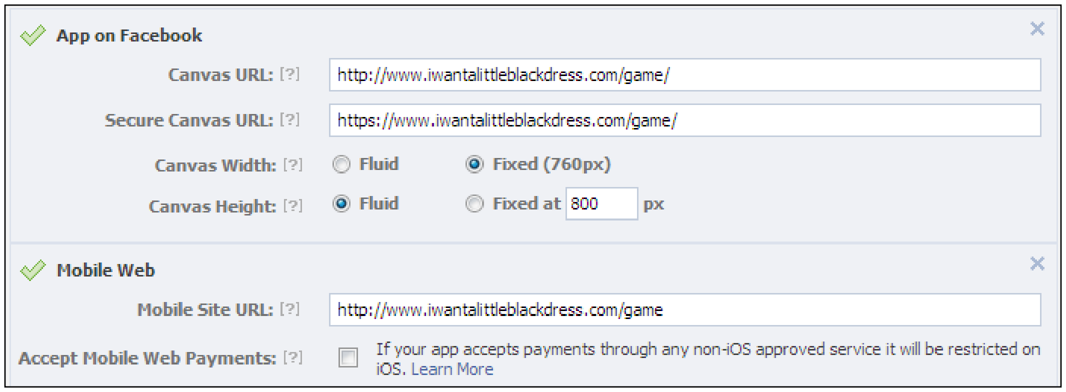
`http://developers.facebook.com`

If you already have your game up and running on your own website, Facebook allows you to set a viewport to the game on your website.

Following are the steps to make your game available on Facebook:

1. In the app section of your account, create a new app by clicking on the following button:

2. Fill out the **App on Facebook** section, as shown in the following screenshot. If your game can also be viewed by mobile devices, you can also fill out the **Mobile Web** section. Make sure **sandbox mode** is on until you have tested everything thoroughly.



App on Facebook

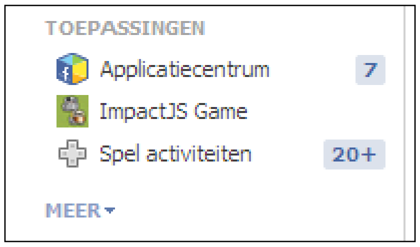
Canvas URL: [?]	http://www.iwantalittleblackdress.com/game/	
Secure Canvas URL: [?]	https://www.iwantalittleblackdress.com/game/	
Canvas Width: [?]	Fluid	[✓ Fixed (760px)]
Canvas Height: [?]	[✓ Fluid]	Fixed at 800 px

Mobile Web

Mobile Site URL: [?]	http://www.iwantalittleblackdress.com/game
Accept Mobile Web Payments: [?]	☐ If your app accepts payments through any non-iOS approved service it will be restricted on iOS. Learn More

3. Go to the **App Details** page where you need to fill out some basic information about your game: category, description, and some screenshots. Once you are ready, you can preview your game by clicking on one of the following buttons:

4. Try going back to your own profile page and you will find your game in your applications list, as shown in the following screenshot. Click on it to play and beta test your own Facebook game.



This is not the only way to send your game to Facebook. You can make an actual Facebook app using AppMobi. However, once your game is finished and somewhere on a web server, this is a very quick way to get it on Facebook. There is also a big advantage to this method: the game remains on a server controlled by you, Facebook merely provides a viewport. This means that if Facebook changes things, this has little or no effect on compatibility with your game and you will not be pressed to change code anywhere.

As a last topic of this chapter, we will have a quick look at the direct canvas implementation of AppMobi. This is an interesting concept since it allows for games to run much faster than they would otherwise. However, first let's recap.

Summing up the preceding content, the conclusions are as follows:

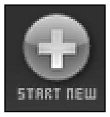
- There are several ways to bring your game to Facebook. Since we have already used AppMobi to build apps, we investigate the viewport solution.
- You need your game to be up and running on a server and a free Facebook developers account.
- Go to the **App** section and create a new app with normal canvas and/or mobile URL. Also fill out all the app details.
- Beta test your game thoroughly before releasing it. You can find your game between the applications in your own personal Facebook page.

Direct canvas game acceleration with AppMobi

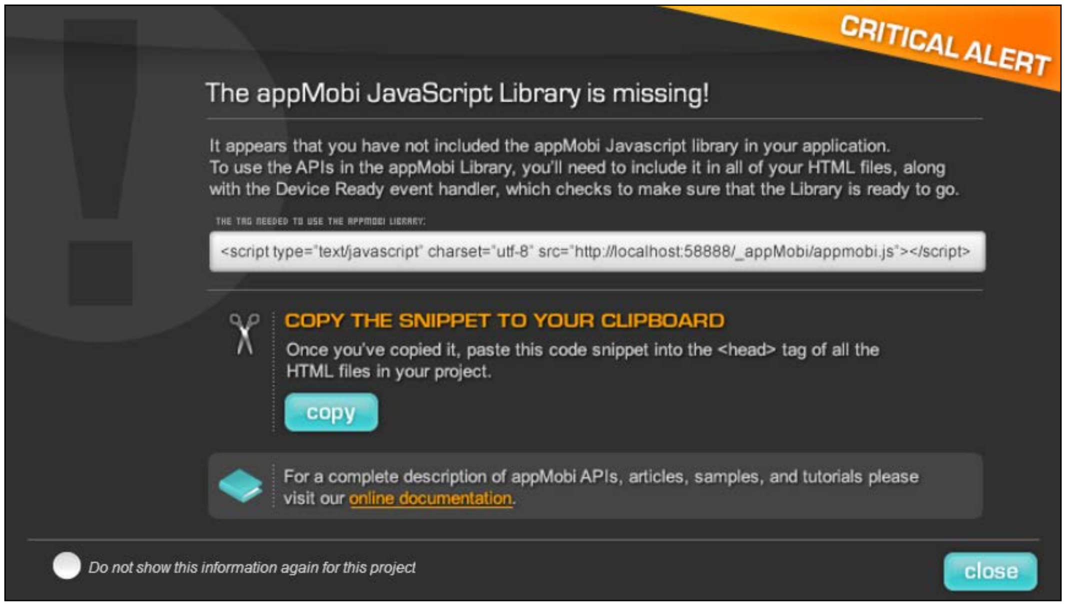
HTML5 games are great because the HTML and JavaScript can be interpreted by any browser and conversion to an app is rather straightforward. The easy "deployability" is a great advantage, but it comes with a rather big disadvantage. The resources taken by the canvas element in order to actually render your game can be outrageous, easily resulting in system lag once you want to use many entities at once. There are few things worse to the game experience than such a large drop in frame rate, it becomes like watching a slide show. However, there are tricks to improving this such as prerendering the graphics as suggested in *Chapter 7*, *Graphics*.

If you want to make use of the performance boost that direct canvas provides, the implementation is rather straightforward. However, first you will need to prepare your code for the AppMobi ImpactJS XDK. Following are the steps to implement direct canvas acceleration:

1. Go to the Chrome Web Store and install the Impact XDK extension.
2. In the XDK, log in to your AppMobi account and add a new project. Choose your game's `root` folder in your XAMPP (or WAMP) library. The following screenshot shows the button to start a new project:



3. The XDK will warn you that you have not yet included the AppMobi library in your game and you will not be able to make use of the AppMobi commands. Copy the code shown in the following screenshot to your clipboard, as suggested by the pop-up:



The appMobi JavaScript Library is missing!

It appears that you have not included the appMobi Javascript library in your application. To use the APIs in the appMobi Library, you'll need to include it in all of your HTML files, along with the Device Ready event handler, which checks to make sure that the Library is ready to go.

THE TAG NEEDED TO USE THE APPMOBI LIBRARY:

```
<script type="text/javascript" charset="utf-8" src="http://localhost:58888/_appMobi/appmobi.js"></script>
```

COPY THE SNIPPET TO YOUR CLIPBOARD

Once you've copied it, paste this code snippet into the <head> tag of all the HTML files in your project.

For a complete description of appMobi APIs, articles, samples, and tutorials please visit our online documentation.

Do not show this information again for this project

4. Open your `index.html` file and paste the script in the `head` section of the document. Your game is now ready to be viewed in the Impact XDK and you can add AppMobi commands to it when necessary, as shown in the following code snippet:

```
<!-- the line below is required for access to the appMobi JS
library -->
<script type="text/javascript" charset="utf-8" src="http://
localhost:58888/_appMobi/appmobi.js"></script>
<script type="text/javascript" language="javascript">
  // This event handler is fired once the AppMobilibraries are
ready
  function onDeviceReady() {
    //use AppMobi viewport to handle device resolution differences
if you want
    //AppMobi.display.useViewport(768,1024);
    //hide splash screen now that our app is ready to run
    AppMobi.device.hideSplashScreen();
  }
  //initial event handler to detect when appMobi is ready to roll
  document.addEventListener("appMobi.device.
ready",onDeviceReady,false);
</script>
```

We now have our game running in the XDK. However, we do not yet have direct canvas acceleration.

5. Create a new script called `index.js` in your game's `root` folder and add the following code:

```
AppMobi.context.include( 'lib/impact/impact.js' );
AppMobi.context.include( 'lib/game/main.js' );
```

6. Open `index.html` and add the `AppMobi` command to the `onDeviceReady()` event listener. The following code will load the `index.js` script:

```
functiononDeviceReady() {
  AppMobi.device.hideSplashScreen();
  AppMobi.canvas.load("index.js");
}
```

7. Remove the following `script` tags that include your game and impact engine scripts:

```
<script type="text/javascript" src="lib/impact/impact.js"></
script>
<script type="text/javascript" src="lib/game/main.js"></script>
```

8. Remove the following `canvas` element:

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

9. Open `main.js` script and add the following to the list of required scripts:

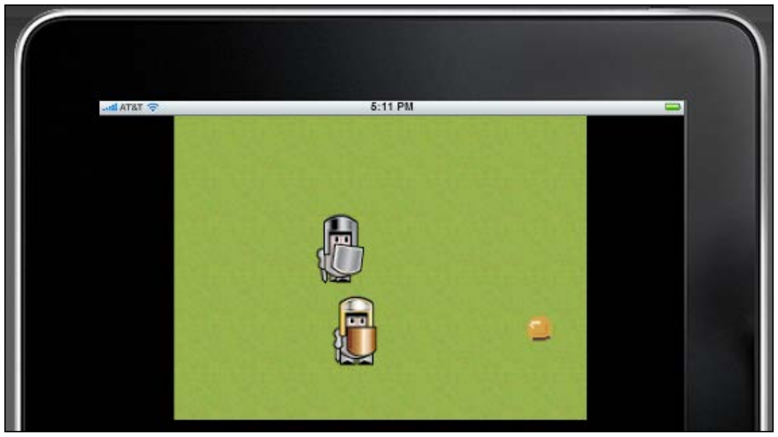
```
'plugins.dc.dc'
```

10. If you have any reference to canvas styling, remove it from your code. For example: `ig.system.canvas.style.width = '320px'`.

11. Finally, remove the touch event bindings you might have implemented when preparing the touch events for mobile and replace them with the AppMobi versions. The `<div>` elements can stay in the `index.html` file, but you need to attach other events. For instance, for the `shoot` button`<div>` element:

```
onTouchStart="AppMobi.canvas.execute('ig.input.
actions[\'shoot\']=true;ig.input.presses[\'shoot\']=true;');"
onTouchEnd="AppMobi.canvas.execute('ig.input.delayedKeyup.push(
\'shoot\' )');"
```

Congratulations! You have now successfully implemented direct canvas acceleration! You might notice that the canvas element outline is now gone when playing your game in the Impact XDK, as shown in the following screenshot:



Summary

The goal of this chapter was to provide insight into the technical preparations necessary when publishing your game in several ways. First we took a look at the process of baking your game code, which allows for shorter loading times and makes the source code less readable. Baking should be done right before distributing any game. Then we went deeper into adapting your game for mobile browsers by implementing touch controls. Turning the game into a web app or an Android app was done with the help of AppMobi. When publishing to Facebook, you have several options and we took a deeper look at one of them. In this solution, your own website acts as the actual platform while Facebook merely provides a viewport. On mobile devices, processing power and memory usage can be real issues when running a game. Therefore, we took a look at the direct canvas implementation of AppMobi. By getting rid of the normal HTML canvas element, a lot of overhead processing can be eliminated, greatly reducing the necessary resources.

In the next and final chapter, we will have a look at the options you have as an HTML5 game developer to make money and hopefully turn a hobby into your job.

9
Making Money with Your Game

In this final chapter we will take a quick look at the options you have for making money with HTML5 game development. Building games can be done purely as a hobby or as a profession. However, the latter requires you to build some pretty unique and successful games as the competition is quite steep. Thus, offering a unique gaming proposition, supported by a healthy dose of marketing, seems to be the way to go for most successful game developers. In this chapter we will cover:

- A few strategic options you have when going into game development
- Making money in the app circuit of Android and Apple
- The option of in-game advertising and how it applies to HTML5 games
- MarketJS as a way to sell your distribution rights to a publisher

Your game development strategy

If you want to build a game to make some money, it is imperative you take a few things into consideration before starting off building one. The first question you will need to ask yourself is probably this: who am I going to build a game for? Are you aiming at everyone capable of playing games or do you want to target a very specific segment of people and meet their gaming needs? This is the difference between broad and niche targeting. An example of very broadly targeted games are most tower defense games, in which you need to build towers with diverse properties to repel an army. Games such as Tetris, Bejeweled, Minesweeper, and most light puzzle games in general. Angry Birds is another example of a game that is popular with a broad audience because of its simplicity, likeable graphics, and incredible amount of clever marketing.

Casual games in general seem to appeal to the masses because of the following few factors:

- Simplicity prevails: most gamers get used to the game in mere minutes.
- There are little or no knowledge prerequisites: you are not expected to already know some of the background story or have experience in these types of games.
- Casual gamers tend to do well even though they put in less time to practice. Even if you do well from the start, you can still become better at it. A game at which you cannot become better by replaying doesn't hold up for long. Notable exceptions are games of chance like roulette and the slots, which do prove to be addictive; but that is for other reasons, such as the chance to win money.

The main advantage to building casual games is that practically everyone is a potential user of your game. As such, the achievable success can be enormous. World of Warcraft is a game which has moved from rather hardcore and niche to more casual over the years. They did this because they had already reached most regular gamers out there, and decided to convince the masses that you can play World of Warcraft even if you don't play a lot in general. The downside of trying to please everyone is the amount of competition. Sticking out as a unique game among numerous games out there is tremendously difficult. This is especially true if you don't have an impressive marketing machine to back it up.

A good example of a niche game is any game built after a movie. Games on Star Trek, Star Wars, Lord of the Rings, and so on, are mostly aimed at people who have seen and liked those movies. Niche games can also be niche because they are targeted solely at a specific group of gamers. For example, people who prefer playing FPS (First Person Shooters) games, and do so every single day. In essence, niche games have the following properties (note that they oppose casual or broadly targeted games):

- Steep learning curve: mastery requires many hours of dedicated gaming.
- Some knowledge or experience with games is required. An online shooter game such as PlanetSide 2 requires you to have at least some experience with shooter games in the past, since you are pitted against people who know what they are doing.
- The more you play the game, the more are the useful rewards you get. Playing a lot is often rewarded with items that make you even stronger in the game, thus reinforcing the fact that you already became better by playing more.

StarCraft is a game released by Blizzard in 1998 and is still being played in tournaments today, even though there is a follow up: StarCraft 2. Games such as the original StarCraft are perfectly feasible to be built in HTML5 and run on a browser or smartphone. When StarCraft was released, the average desktop computer had less power than many smartphones have today. Technically, the road is open; replicating the same level of success is another matter though.

The advantage of aiming at a niche of gamers is the unique position you can take in their lives. Maybe there are not many people in your target group, but it will be easier to get and keep their attention with your game since it is specifically built for them. In addition, it doesn't mean that because you have a well-defined aim, gamers can't drop in from unexpected corners. People you would have never thought would play your game can still like what you did. It is for this reason that knowing your gamers is so important, and why tools such as Playtomic exist.

The disadvantage of the niche marketing is obvious: your game is very unlikely to grow beyond a certain point; it will probably never rank among the most played games in the world.

The type of games you are going to develop is one choice, the amount of detail you put in each one of them is another. You can put as much effort into building a single game as you want. In essence, a game is never finished. A game can always have an extra level, Easter egg, or another nice little detail. When scoping your game, you must have decided whether you will use a shotgun or a sniper development strategy.

In a shotgun strategy, you develop and release games quickly. Each game still has a unique element that should distinguish it from other games: the UGP (Unique Gaming Proposition). But games released under a shotgun strategy don't deliver a lot of details; they are not polished.

The advantages of adopting a shotgun strategy are plenty:

- Low development cost; thus every game represents a low risk
- The short time to market allows for using world events as game settings
- You have several games on the market at once, but often you only need a single one to succeed in order to cover the expenses incurred for the others
- Short games can be given to the public for free but monetized by selling add-ons, such as levels

However, it's not just rainbows and sunshine when you adopt this strategy. There are several reasons why you wouldn't choose shotgun strategy:

- A game that doesn't feel finished has less chance of being successful than a perfected one.
- Throwing your game on the market not only tests whether a certain concept works, but also exposes it to the competitors, who can now start building a copy. Of course, you have the first mover advantage, but it's not as big as it could have been.
- You must always be careful not to throw garbage on the market either, or you might ruin your name as a developer.

However, don't get confused. The shotgun strategy is not an excuse for building mediocre games. Every game you release should have an original touch to it—something that no other game has. If a game doesn't have that new flavor, why would anyone prefer it over all the others?

Then, of course, there is the sniper strategy, which involves building a decent and well-thought-out game and releasing it to the market with the utmost care and support. This is what distributors such as Apple urge developers to do, and for good reason—you wouldn't want your app store full of crappy games, would you? Some other game distributers, such as Steam, are even pickier in the games they allow to be distributed, making the shotgun strategy nearly impossible. But it is also the strategy most successful game developers use. Take a look at developers such as Rockstar (developer of the GTA series), Besthesda (developer of the Elder Scroll series), Bioware (developer of the Mass Effect series), Blizzard (developer of the Warcraft series), and many others. These are no small fries, yet they don't have that many games on the market. This tactic of developing high quality games and hoping they will pay off is obviously not without risk. In order to develop a truly amazing game, you also need the time and money to do so. If your game fails to sell, this can be a real problem for you or your company. Even for HTML5 games, this can be the case, especially since devices and browsers keep getting more and more powerful. When the machines running the games get more powerful, the games themselves often become more complex and take longer to develop.

We have taken a look at two important choices one needs to make when going into the game-developing business. Let's now look at the distribution channels that allow you to make money with your game, but before that let's summarize the topic we have just covered:

- Deciding who you want to target is extremely important even before you start developing your game.
- Broad targeting is barely targeting at all. It is about making a game accessible and likeable by as many people as possible.
- Niche targeting is taking a deeper look and interest in a certain group of people and building a game to suit their specific gaming needs.
- In developing and releasing games, there are two big strategies: shotgun and sniper.
- In the shotgun strategy you release games rapidly. Each game still has unique elements that no other games possess, but they are not as elaborate or polished as they could be.
- With the sniper strategy you build only a few games, but each one is already perfected at the time of release and only needs slight polishing when patches are released for it.

Making money with game apps

If you built your game into an app, you have several distribution channels you can turn to, such as Firefox marketplace, the IntelAppUp center, Windows Phone Store, Amazon Appstore, SlideMe, Mobango, Getjar, and Apple Appsfire. But the most popular players on the market currently are **Google Play** and the **iOS App Store**. The iOS App Store is not to be confused with the Mac app store. iPad and Mac have two different operating systems, iOS and Mac OS, so they have separate stores. Games can be released both in the iOS and the Mac store. There could also be some confusion between Google Play and **Chrome Web Store**. Google Play contains all the apps available for smartphones that have Google's Android operating system. The Chrome Web Store lets you add apps to your Google Chrome browser. So there are quite a few distribution channels to pick from and here we will have a quick look at Google Play, the iOS App store, and the Chrome Web Store.

Google Play

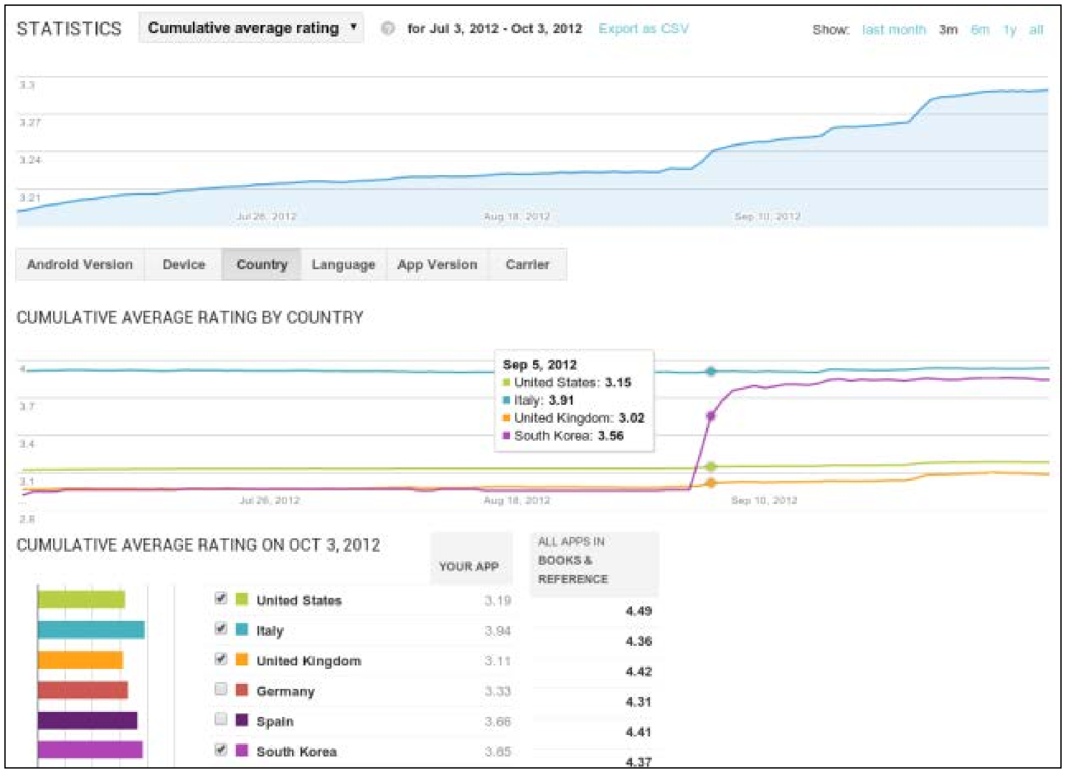
Google Play is the Android's default app shop and the biggest competitor of the iOS App Store.

If you wish to become an Android app developer, there is a $25 fee and a developer distribution agreement that you must read.

In return for the entrance fee and signing this agreement, they allow you to make use of their virtual shelves and have all its benefits. You can set your price as you please, but for every game you sell, Google will cash in about 30 percent. It is possible to do some geological price discrimination. So you could set your price at, let's say €1 in Belgium, while charging €2 in Germany. You can change your price at any time; however, if you release a game for free, there is no turning back. The only way to monetize this app afterwards is by allowing in game advertising, selling add-ons, or creating an in-game currency that can be bought with real money.

Introducing an in-game currency that is bought with real money can be a very appealing format. A really successful example of this monetizing scheme can be found in The Smurfs games. In this game you build your own Smurf village, complete with Big Smurf, Smurfette, and a whole lot of mushrooms. Your city gets bigger as you plant more crops and build new houses, but it is a slow process. To speed it up a bit you can buy special berries in exchange for real money, which in turn allows you to build exclusive mushrooms and other things. This monetization scheme becomes very popular as has been shown in games such as League Of Legends, PlanetSide 2, World of Tanks, and many others. For Google Play apps, this in-app payment system is supported by Android's Google Checkout.

In addition, Google allows you access to some basic statistics for your game, such as the number of players and the devices on which they play, as shown in the following diagram:



CUMULATIVE AVERAGE RATING ON OCT 3, 2012

	YOUR APP	ALL APPS IN BOOKS & REFERENCE
✓ United States	3.19	4.49
✓ Italy	3.94	4.36
✓ United Kingdom	3.11	4.42
Germany	3.33	4.31
Spain	3.66	4.41
✓ South Korea	3.65	4.37

Information such as this allows you to redesign your game to boost your success. For example, you could notice if a certain device doesn't have that many unique users, even though it is a very popular device and is bought by many people. If this is the case, maybe your game doesn't looks nice on this particular smartphone or tablet and you should optimize for it.

The biggest competitor and initiator of all apps is the iOS App Store, so let's have a look at this.

iOS App Store

The iOS App Store was the first of its kind and at the time of writing this book, it still has the biggest revenue.

In order to publish apps in the iOS App Store, you need to subscribe to the iOS Developer Program, which costs $99 annually—almost four times the subscription fee of Google Play. In effect, they offer about the same thing as Google Play does; as you can see in this short list:

- You pick your own prices and get 70 percent of sales revenue
- You receive monthly payments without credit card, hosting, or marketing fees
- There is support and adequate documentation to get you started

More importantly, here are the following differences between Google Play and the iOS App Store:

- As mentioned earlier, signing up for Google Play is cheaper.
- The screening process of Apple seems to be more strict than that of Google Play, which results in a longer time to reach the market and a higher chance of never even reaching the market.
- Google Play incorporates a refund option that allows the buyer of your app to be refunded if he or she uninstalls the app or game within 24 hours.
- If you want your game to make use of some Android core functionalities, this is possible since the platform is open source. Apple, on the other hand, is very protective of its iOS platform and doesn't allow the same level of flexibility for apps. This element might not seem that important for games just yet, but it might be for very innovative games that do want to make use of this freedom.
- iOS reaches more people than Android does, though the current trend indicates that this might change in the near future.
- There seem to be significant differences in the kind of people buying Apple devices and the users of smartphones or tablets with Android OS. Apple fans tend to have a lower barrier towards spending money on their apps than Android users do. In general, iPads and iPhones are more expensive than other tablets and smartphones, attracting people who have no problem with spending even more money on the device. This difference in target group seems to make it more difficult for Android game developers to make money from their games.

> If your game works on a Safari browser, it does not mean it is ready to be accepted in the iOS App Store. Turning your game into a native app takes a bit of extra preparation. The same goes for the Chrome browser and the Google Play store. The conversion from browser game to app can be done using AppMobi, as has been shown in *Chapter 8, Adapting Your HTML5 Game to the Distribution Channels*.

The last option for selling apps that we will discuss here is the Chrome Web Store.

The Chrome Web Store

The Chrome Web Store differs from Google Play and the iOS App Store in that it provides apps specifically for the Chrome browser and not for mobile devices.

The Chrome Store offers web apps. Web apps are like applications you would install on your PC, except web apps are installed in your browser and are mostly written using HTML, CSS, and JavaScript, like our ImpactJS games. The first thing worth noticing about the Chrome Store is the one-time $5 entrance fee for posting apps. If this in itself is not good enough, the transaction fee for selling an app is only 5 percent. This is a remarkable difference to both Google Play and the iOS App Store. If you are already developing a game for your own website and packaged it as an app for Android and/or Apple, you can just as well launch it on the Chrome Web Store. Turning your ImpactJS game into a web app for the Chrome Store can be done using AppMobi, yet Google itself provides detailed documentation on how to do this manually.

One of the biggest benefits of the web app is the facilitation of the permission process. Let's say your web app needs the location of the user in order to work. While iPad apps ask for permission every time they need location data, the web app asks permission only once: at installation time.

Furthermore, you have the same functionalities and payment modalities as in Google Play, give or take. For instance, there is also the option to incorporate a free trial version, also known as a freemium. A freemium model is when you allow downloading a demo version for free with the option of upgrading it to a full version for a price. The Smurfs game also uses the freemium model, albeit with a difference. The entire game is free, but players can opt to pay real money to buy things that would otherwise cost them a lot of time to acquire. In this freemium model, you pay for convenience and unique items. For instance, in PlanetSide 2, acquiring a certain sniper rifle might take you several days or $10, depending on how you choose to play the freemium game.

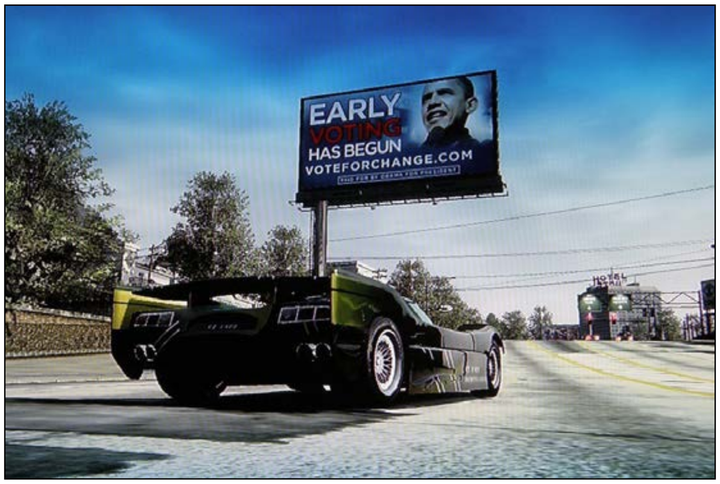
If you plan on releasing an ImpactJS game for Android, there is no real reason why you wouldn't do so for the Chrome Web Store.

That being said, let's have a quick recap:

- The time when the iOS app store was the only app store out there is long gone; there is an impressive repertoire of app stores to choose from, which includes Firefox Marketplace, Intel AppUp Center, Windows Phone Store, Amazon Appstore, SlideMe, Mobango, Getjar, Appsfire, Google Play, among others.
- The biggest app stores are currently Google Play and the iOS App Store. They differ greatly on several fronts, of which the most important ones are:
    - Subscription fee
    - Screening process
    - Type of audience they attract
- The Chrome Web Store sells web apps that act like normal apps but are available in the Chrome browser.
- The Chrome Web Store is cheap and easy to subscribe to. You must definitely have a go at releasing your game on this platform.

In-game advertising

In-game advertising is another way to make money with your game. In-game advertising is a growing market and is currently already being used by major companies; it was also used by Barack Obama in both his 2008 and 2012 campaigns, as shown in the following in-game screenshot:



There is a trend towards more dynamic in-game advertising. The game manufacturers make sure there is space for advertising in the game, but the actual ads themselves are decided upon later. Depending on what is known about you, these can then change to become relevant to you as a player and a real-life consumer.

When just starting out to build games, in-game adverting doesn't get that spectacular though. Most of the best known in-game advertisers for online games don't even want their ads in startup games.

The requirements for Google AdSense are the following:

- **Game plays**: Minimum 500,000 per day
- **Game types**: Web-based Flash only
- **Integration**: Must be technically capable of SDK integration
- **Traffic source**: Eighty percent of traffic must be from the US and the UK
- **Content**: Family-safe and targeted at users aged 13 and over
- **Distribution**: Must be able to report embedded destinations and have control over where the games are distributed

The requirements for another big competitor, Ad4Game, aren't mild either:

- At least 10,000 daily unique visitors
- Sub-domains and blogs are not accepted
- The Alexa rank should be less than 400,000
- Adult/violent/racist content is not allowed

If you are just starting out, these prerequisites are not good news. Not only because you need such a large number of players before even starting the advertising, but also because currently all support goes to Flash games. HTML5 games are not fully supported yet, though that will probably change.

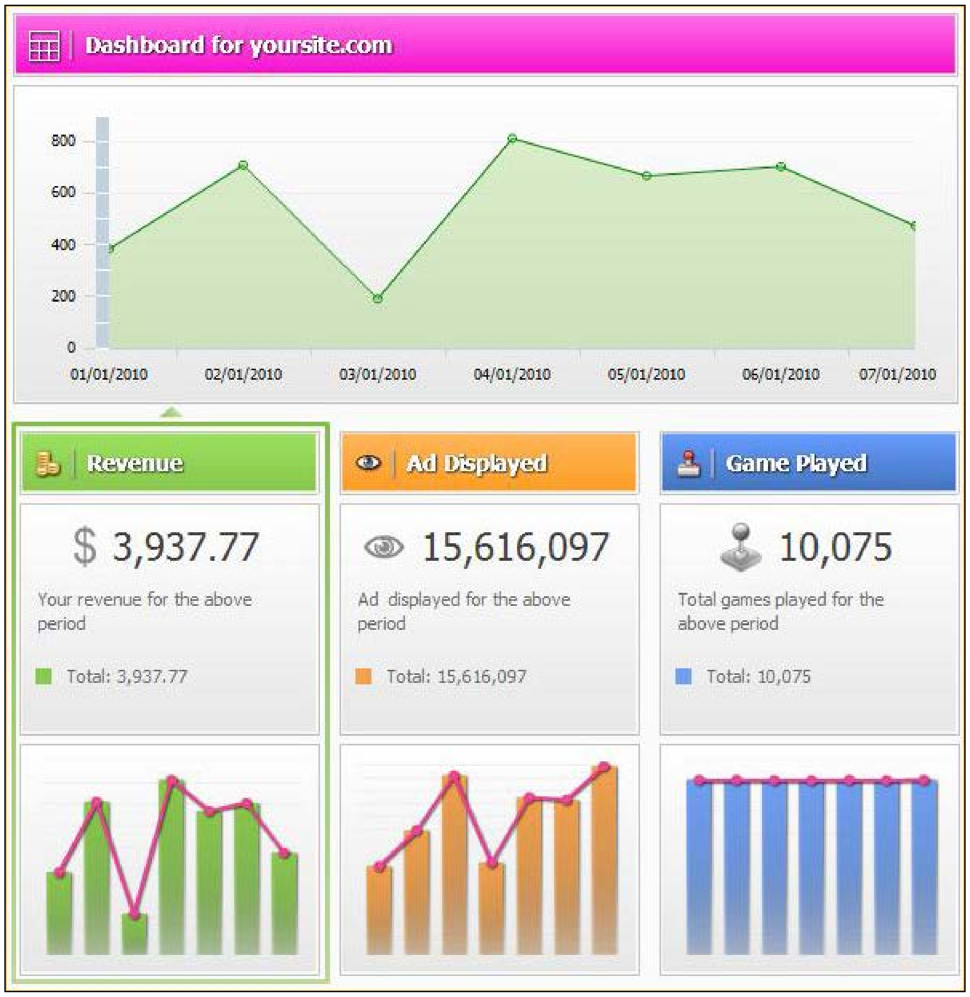
Luckily, there are companies out there that do allow you to start using advertising even though you don't have 10,000 visitors a day. Tictacti is one of those companies.

Once again, almost all support goes to the Flash games, but they do have one option available for an HTML5 game: **pre-roll**. Pre-roll simply means that a screen with an ad appears before you can start the game. Integration of the pre-roll advertising is rather straightforward and does not require a change to your game, but to your `index.html` file, as in the following example from Tictacti:

```
//You can use publisherId 3140 and tagTypedemoAPI for testing purposes
however the ads will not be credit to you.
<html>
<head>
  <title>Simple Ad</title>
</head>
<body>
<script type="text/javascript"
src="http://cdn.tictacti.com/widgets/js/t3widgets.js"></script>
<script type="text/javascript">
  var publisherId = "3140";
  var tagType = "jsGameAPI";
  var agencyUniqueId= "0";
  var playerWidth = "600";//The Game width
  var playerHeight = "400";//The Game height
  var t3cfg = {
    wrapperUrl: 'engine/game/3170/tttGameWrapper.swf',
config: { enableDM: false, tttPreloader: false, bgcolor: "#000000"
, engineConnectorType: 7 , externalId:agencyUniqueId},
    onClose:
function(){document.location="http://www.tictacti.com";}
    //Called after the ad is closed. In the Demo after 30 seconds.
  };
  TicTacTi.renderWidget(publisherId, tagType, playerWidth ,
playerHeight , t3cfg);
</script>
</body>
</html>
```

While adding this to your game's `index.html` file, you fill out your own publisher ID and you are basically ready to go.

Tictacti is similar to Google Analytics, and it also provides you with some relevant information about the ads on your game's website, as shown in the following diagram:



$ 3,937.77

Your revenue for the above period

🟩 Total: 3,937.77

15,616,097

Ad displayed for the above period

🟧 Total: 15,616,097

10,075

Total games played for the above period

🟦 Total: 10,075

Be careful, however, pre-roll advertising is one of the most intruding and annoying kinds of advertising. Technically, it is not even in-game advertising at all, since it runs before you play the game. If your game is not yet well established enough to convince the player to endure the advertising before being able to play, don't choose this option. Give your game some time to build a reputation before putting your gamers through this.

As the last option, we will have a look at selling your actual distribution rights with MarketJS. But let's first briefly recap on in-game advertising:

- In-game advertising is a growing market. Even Barack Obama made use of in-game billboards to support his campaign.
- There is a trend towards more dynamic in-game advertising—using your location and demographic information to adapt the ads in a game.
- Currently, even the most accessible companies that offer online in-game advertising are focused on Flash games and require many unique visitors before even allowing you to show their ads. Tictacti is a notable exception, as it has low prerequisites and an easy implementation; though advertising is currently limited to pre-roll ads.
- Always take care to first build a positive reputation for your game and allow advertisements later.

Selling distribution rights with MarketJS

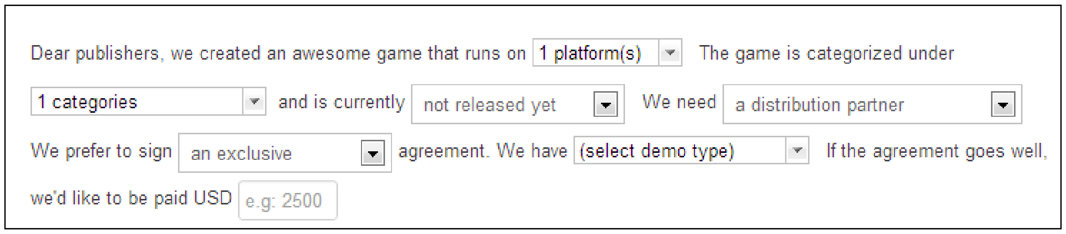
The last option we will investigate in this chapter is selling your game's distribution rights. You can still make money by just posting your game to all the app stores and on your own website, but it becomes increasingly difficult to be noticed. Quality can only prevail if people know it is out there, thus making a good game is sometimes not enough—you need marketing. If you are a beginner game builder with great game ideas and the skills to back it up, that's great, but marketing may not be your cup of tea. This is where MarketJS comes into play.

MarketJS acts as an intermediate between you as the game developer and the game publishers.

The procedure is simple once you have a game:

1. You sign up on their website, `http://www.marketjs.com`.
2. Upload the game on your own website or directly to the MarketJS server.
3. Post your game for publishers to see. You set several options such as the price and contract type that would suit you the best. You have five contract options:
    - **Complete distribtution contract**: Sell all your distribution rights to the game.
    - **Exclusive distribution partner contract**: Here you restrict yourself to work with one distributor but still retain the rights to the game.
    - **Non-exclusive contract**: Here, any distributor can buy the rights to use your game, but you can go on selling rights as long as you want.

- **Revenue share**: Here you negotiate on how to split the revenues derived from the game.
- **Customized contract**: This can basically have any terms. You can choose this option if you are not sure yet what you want out of your game. A part of the webpage on which you fill out your contracting preferences is shown in the following screenshot:

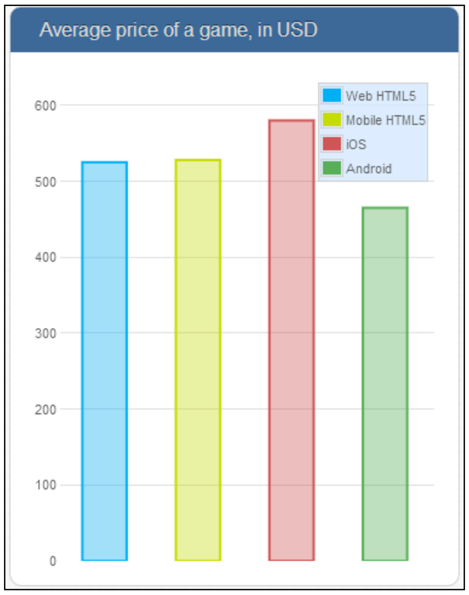


Dear publishers, we created an awesome game that runs on [1 platform(s)] The game is categorized under [1 categories] and is currently [not released yet] We need [a distribution partner] We prefer to sign [an exclusive] agreement. We have [(select demo type)] If the agreement goes well, we'd like to be paid USD [e.g: 2500]

After you have posted a demo, it is a matter of waiting for a publisher to spot it, get stunned by its magnificence, and offer to work with you.

The big contribution of MarketJS to the gaming field is this ability to let the game developer focus on developing games. Someone else takes care of the marketing aspect, which is a totally different ballgame.

MarketJS also offers a few interesting statistics such as the average price of a game on their website, as shown in the following diagram. It grants you some insight on whether you should take up game developing as a living or keep doing it as a hobby.



According to MarketJS, prices for non-exclusive rights average between $500 and $1000, while selling exclusive rights to a game range somewhere between $1500 and $2000. If you can build a decent game within this price range, you are more than ready to go:

- MarketJS is a company that brings game distributers and developers closer together. Their focus is on HTML5 games, so they are great if you are a startup ImpactJS game developer.
- They require no subscription fee and have a straightforward process to turn your game into a showcase with a price tag.

Summary

In this chapter we have taken a look at some important elements while considering your game development strategy. Do you wish to adapt a shotgun approach and develop a lot of games in a short time span? Or will you use the sniper strategy and only build a few, but very polished games? You also need to decide upon the audience you wish to reach out to with your game. You have the option of building a game that is liked by everyone, but the competition is steep.

Making money on the application stores is possible, but for Android and Apple there are registration fees. If you decide to develop apps, it is worth giving the Chrome Web Store a try (it runs web apps).

In-game advertising is another way to fund your efforts, though most companies offering this service for online games have high prerequisites and support Flash games more than they do the newer HTML5 games.

One of the most promising of monetization schemes is the freemium model. Players are allowed to freely play your game but they pay real money for extras. This is an easily tolerated model since the game is essentially free to play for anyone not willing to spend money, and no annoying advertising is present either.

A combination of in-game advertising and freemium is possible as well: people annoyed by the advertising pay a fee and in return they are not bothered by it anymore.

A final option is leaving the marketing aspect to someone else by selling your distribution rights with the help of MarketJS. They aim for HTML5 games and this option is especially useful for the beginner game developer who has difficulty in marketing his or her game.

We have now reached the end of the book and quite a lot of information has been covered – from the basics of setting up your server, over the development of an ImpactJS game, and to the distribution of your own creation. I thank you for reading all of this.

My hope was to make this entire process a bit more graspable and deliver the final push for you to start creating your own games and maybe even make a living out of it. Sometimes, developing a game can be frustrating, as the devil is often in the detail. You might find yourself cursing your screen more often than you like, but remember that it's not the computer's fault, and with enough determination you will always find a solution. If you find yourself lost at times, the ImpactJS website has a great forum full of very helpful people and I would greatly encourage you to make use of it and share your thoughts, questions, and ideas there. When you close this book and start building your game, don't forget that having a plan and working in a very structured way will be the key elements to success and might prevent many sleepless nights. Making changes and improvements in a step-by-step manner and always checking if everything still works as intended is the way to go and deliver a perfectly functioning game. But regardless of all the benefits that can be reaped from organized and structured thinking, there lies another most important thing: your imagination.

Think original, don't be bound by what is already out there, and you will no doubt create a game that millions of people might enjoy and, who knows, will even stand the test of time.

Index

Symbols

<div> element 254

A

acceleration, objects
  setting up 56
action array 94
Ad4Game
  requisites 267
addAnim() method 59, 60, 88
addCoin() method 102
addOption() function 163
addProjectile() function 105
Adobe Photoshop
  about 230
  using 230
afterRun() function 164
AI
  about 92, 93
  Data driven AI 93
  single strategy AI 93
AI, side scroller game
  about 138
  adding 138-140
AMPP 8
Androids Google play store
  HTML game, pushing 247, 249
Angry Birds game 257
AnimationSheet() method 59
animation sheets
  used, for enhancing game graphics 59
animSheet property 88
Apache 8
API key 29, 248
API URL 29
Apphub 26
ApplyForce() method 125
ApplyImpulse() function 131
AppMobi
  about 10, 25, 243
  Apphub 26
  URL 10, 25, 243
  used, for implementing direct canvas game acceleration 251, 253
AppMobi appHub 247
AppMobi applab 26
AppMobi command 253
AppMobi ImpactJS XDK 252
Artificial Intelligence. *See* **AI**
avatar
  creating, Adobe Photoshop used 230, 231

B

background music
  adding, to game 216, 217
balloon entity, RPG 110
basic conversation, RPG
  building 185-192
basic tune
  creating, FL Studio used 210-215
beforeRun() function 164
Bejeweled 257
Besthesda 260
Bioware 260
Blizzard 260
bomb entity, side scroller game
  building 133-137
Borderlands 74
Boss entity, RPG 1153

boss entity, side scroller game
creating 148
Box2D
side scroller game, preparing for 119
used, for game physics 68

C

camera view
about 63
automatically following camera 65
free to move camera 64
game canvas 63, 64
canvas properties
fillRect() 195
fillstyle 194
fillText() 195
font 195
setAlpha() 195
cartoonizer effect 231
casual games
advantage 258
examples 257
features 258
changechoices() method 190
checkCookie() function 169
check() function 91
chosen_reply_key variable 190
Chrome debugger
about 16
Audits tab 16
Console tab 16
Elements tab 16
exerises 17-21
Network tab 16
Profiles tab 16
Resources tab 16
Sources tab 16
Timeline tab 16
Chrome Web Store 261, 265
coin entity, RPG 102
CollisionAt() funtion 182
collision.js script 119
collision layer
about 41
dimensions 36
distance 37
tileset, creating 36
tilesets 36
tilesize 36
collision.trace() method 99
complete distribtuin contract 270
composers 207
Console.assert() function 160
console.log() function 160
contract options, MarketJS
complete distribtuin contract 270
customized contract 271
exclusive distribution partner contract 270
non-exclusive contract 270
revenue share 271
cookies
used, for storing data on player's computer 168-172
Cordova 27
creation
adding, to RPG 232
CSS (Cascading Style Sheets) code 238
currentAnim property 88
custom debug panel
adding, to ImpactJS debugger 161-164
customized contract 271

D

data coding 29
Data driven AI 93
debugging, ImpactJS game
Chrome debugger 16, 17
Firebug 15
with browser 15
with ImpactJS 22-24
debugging options 159
DebugPanel 162
defeatable opponent
adding, to RPG 84-86
deployability 251
direct canvas game acceleration
about 26
implementing, with AppMobi 251-254
distancetotargetx variable 180
distancetotargety variable 180
distribution channels
Amazon Appstore 261

Apple Appsfire 261
Firefox 261
Getjar 261
Google Play 261
IntelAppUp center 261
iOS App Store 261
Mobango 261
SlideMe 261
Windows Phone Store 261
doAction() method 100
document.cookie variable 170
Document Object Model (DOM) 15, 143
DOM storage 172
draw() function 155, 176
draw() method 110
Dropbox 249
dummyUnitTest() 159

E

Eclipse 12
Ejecta
about 25
downloading 25
entities
adding, to game 39
entities layer 39
examples, casual games
Angry Birds 257
Bejeweled 257
Minesweeper 257
Tetris 257
examples, niche games
Games on Star Trek 258
Lord of the Rings 258
Star Wars 258
World of Warcraft 258
examples, RPG
Baldur's Gate 74
Final Fantasy 74
Neverwinter Nights 74
Pokémon 74
Tibia 74
World Of Warcraft 74
Zelda 74
examples, side scroller game
2DMario 118
Castlevania 118
Donkey Kong 118
Double Dragon 118
Gameboy Metroid games 118
old Mega Man 118
Sonic 118
Super Nintendo 118
exclusive distribution partner contract 270
explosion() method
about 134
parameters 134
working 135
extend() method 80
extra functionalities, for RPG
about 177
basic conversation 185-191
HUD plugin, adding 193-196
intelligent spawn locations, adding 182, 184
player, moving with mouse click 177-181

F

Fable 75
Facebook
HTML game, making available 249-251
fadeout(time) 217
FileZilla 30
Firebug
about 15
exerises 17-21
FL Studio
about 210
used, for creating basic tune 210-215
FPS (First Person Shooters) 258
freemium model 265
FruityLoop studio. *See* **FL Studio**

G

game analytics
Playtomic, implementing 200-204
game data
adding, to game 165-167
handling 165
local storage 173
storing, lawnchair used 174-177
storing on player's computer, cookies used 168-172

game development strategy
about 257
casual games 258
shotgun strategy 259
sniper strategy 260
GameEnd game class 156
game graphics
about 221
buying 223, 224
creating 222
pixel art 223
vector graphic design 223
vector graphics 224-229
game ID 29
gameinfo array 101
GameInfo class 171
GameInfo.coins function 103
GameInfo.getSavedScore() method 176
GameInfo.saveScore() function 203
GameInfo.score function 103
GameInfo.score class 176
game.min.js script 237
gameOver() function 158, 175
game-over screen
adding 156
game physics, with Box2D
about 68
collision impact and bounciness 70
gravity and force 69, 70
Games on Star Trek 258
game start screen
adding 154, 155
gameWon() function 157, 175
getAction() function 97, 100
getCookie() function 170
getEntitiesAt() function 182
getHealth() function 170
getSavedScore() method 176
getScore() method 175, 176
getStrategy() function 198
getUserName() function 171
Git 27
Google AdSense
requisites 267
Google Chrome Web Store
web app, releasing 243
Google Play 261-263
Google Project ID 248
Google webmaster tool 246
graphics 221
graphics in HTML5
tips 233

H

Head-Up Display plugin. *See* **HUD plugin**
healthpotion entity, RPG 101
hive mind intelligence 197-199
HTML5 local storage
using, without lawnchair application 172
HTML game
direct canvas acceleration, implementing with AppMobi 251-254
distribution channels 261
making available, on Facebook 249-251
preparing, for mobile web browsers 238-242
preparing, for web browser 236, 237
pushing, to Android Google play store 247, 249
turning, to web app for Google Chrome Web Store 243-247
HUD plugin
adding to RPG 193-196

I

IDE 12
ig.assert() function 160
ig.cookie array 172
ig.defines() method 80
ig.game.addCoin() method 103
ig.game.spawnEntity method 61
ig.input.pressed 83
ig.input.state 83
ig.Lawnchair() method 174
ig.module() method 80
ig.requires() method 80
ig.spawnlocations variable 182
ig.system.setGame() function 157
IIS (Internet Information Services) 9
Impact game
background music, adding 216, 217
basic tune, creating FL Studio used 210-215
debugging options 159

debugging, with browser 15
debugging, with browser and ImpactJS 13, 14
debugging, with ImpactJS 22, 23
extra functionalities, for RPG 177
game data, handling 165
game start screen, adding 154, 155
music and sound effects 207
music, buying 208
music, creating 208
player controls, setting up 51-53
sound, playing when action happens 217, 218
Start and Game-over screen 154
tunes and sound effects, buying 208
victory screen, adding 156-158

ImpactJS
acceleration, setting 56
AppMobi 25
camera view 63
Ejecta 25
game physics, with Box2D 68
installing 10, 11
lawnchair 25, 27
levels, building 34
music and sound effects, adding 66
objects 46
PhoneGap 25, 27
Playtomic 25, 29
positioning 55
Scoreoid 25, 28
speed, setting 56
tools 25
unit testing 159, 160
webhost 29

ImpactJS debugger
custom debug panel, adding 161-164

ImpactJS entity
about 46, 47
Void entity 46

increaseScore function 103

in-game advertising
about 266-268
Ad4Game 267
Google AdSense 267
summary 270

init() function 83

Inkscape
about 224
downloading 224

inputnumber variable 160

intelligent spawn locations
adding, to RPG 182, 184

iOS App Store
about 261, 264
versus, Google Play 264

iOSImpact 25

J

JSON 14

K

kill() function 62, 91, 135
komodo edit 12

L

LAMP 8
lastpressed variable 90

lawnchair application
about 27
advantages 27
downloading 174
URL 28
used, for storing data 174-177
working 28

League Of Legends 262
levelchange entity, RPG 107
levelchange entity, side scroller game 145

level, Impact game
building 34
collision layer 41
creating 34, 35
entities and objects, adding 39
entities and objects, removing 40
Layers menu 36
levels, connecting 42-45
z axis 38

level, RPG
building 75-79
collision layer 78
grass layer 76
vegetation_back layer 76

vegetation_front layer 76
levels
connecting 42-44
level, side scroller game
building 120-122
LMMS
about 210
download link 210
load() function 164
LoadGame 155
loadlevel() function 120, 161
local storage. *See* **also DOM storage 27**
Lord of the Rings game 258

M

main.js update function 57
MAMP 9
manifest.json file 245
map transitioning, side scroller game 145-147
map transition, RPG
creating 106, 107
MarketJS
about 270
contract options 270
features 271, 272
URL 270
used, for selling distribution rights 270-272
menu init() function 186
Minecraft 74
Minesweeper 257
minor foe
adding, to side scroller game 127, 128
MMORPGs (Mass Multiplayer Online Role Playing Game) 74
mobile web browsers
HTML game, preparing for 238-242
mousemovement() method 178
mousewalking command 178
movementspeed variable 178
multiple-strategy AI 93
music
buying 208
creating 208
music and sound effects
adding 66
background music, playing 66, 67
sound effects 67, 68
music.play() method 217
music.volume method 217
MyGame 154
MySQL 8

N

niche games
about 258
advantage 259
disadvantage 259
examples 258
properties 258
non-exclusive contract 270
Non Playable Characters. *See* **NPCs**
notepad++ 12
npcreply() method 190
NPCs behaviour, RPG 94-97
NPCs decision making process, RPG 97, 99, 100
NPCs
bringing, to life 93
speech balloon 108-111
talking NPC, adding 112-114

O

object literal notation 14
objects
adding, to game 40
death process 62
health, deducting 62
removing, from game 40
spawning 61
onDeviceReady() event listener 253
one.com webhost
about 30
URL 30
OpenAL 25
OpenGL 25
OpenScreen function 155
OS (operating system) 27
outputnumber variable 160

P

parameters, explosion() method
  blast damage 135
  blastforcex 135
  blastforcey 135
  maximum blast-zone 134
  minimum blast-zone 134
PayPal 27
Perl 8
PhoneGap
  about 27
  URL 27
PHP 8
pickup items, side scroller game
  healing crate, implementing 142, 143
  normal crate, adding 140-142
Pixlr
  about 230
  URL 230
PlanetSide 2 262
playable and non-playable characters
  creating 46-50
playable character
  adding, to RPG 80-83
  adding, to side scroller game 122-126
player controls
  setting up 51-53
player.js init() function 57
player.mousewalking variable 179
player.takemouseinput variable 179
Playtomic
  about 29, 200
  implementing, for game analytics 200-204
  URL 29, 200
positioning, objects 55
premade game
  running 12

R

ready() function 164
receive damage function 97
receiveDamage() method 62
requires() method 85
Resident Evil game 208
revenue share 271
Rockstar 260
Role Playing Game (RPG)
  about 34
  building 73
  coin entity 102, 103
  defeatable opponent, adding 84, 86
  final battle 114
  final Boss entity 114, 115
  items, picking up for player 101
  level, building 75-79
  map transition, creating 106, 107
  playable character, adding 80-83
  player, healing with potion 101, 102
  score, monitoring 103-105
  setting 74, 75
  weapons, adding 87

S

saveScore() function 175, 204
Scoreoid
  about 28
  ingame notifications 28
  leaderboards 28
  player login 28
  URL 29
score, RPG
  monitoring 103-105
score, side scroller game
  monitoring 144, 145
script editors
  Eclipse 12
  komodo edit 12
  notepad++ 12
  selecting 12
  sublime edit 2 12
  Textmate 12
  Xcode 12
setCookie() function 169
setHealth() function 170
setMaxHealth() method 195
setScore() function 175
setUserName() function 171
shotgun strategy
  about 259
  advantages 259
  limitations 260

side scroller game
about 118
artificial intelligence, adding 138-140
boss entity 148
building 117
final battle 148-151
level, building 120, 121
map transitioning 145-147
minor foe, adding 127, 128
pickup items 140
playable character 122-125
preparing, for Box2D 119, 120
score, monitoring 144
setting 118
spawnEntity() function 128
weapons, adding 129
single strategy AI 93
Skyrim 75
sniper strategy 260
sortEntitiesDeferred() method 57
sound
playing, when action happens 217, 218
sound files
tips, for using in game 219
spawnEntity method 90
spawnIf() method 182
speed, objects
setting up 57
sprite
about 58
used, for enhancing game graphics 58, 59
Sprites4games
about 223
URL 223
StarCraft 259
Star Wars 258
store.save() method 175
strategies 93
sublime edit 2 12
submitComplete() function 203
substractprojectile() function 105

T

Talkie 113
Tetris 257
textballoon entity, RPG 108
Textmate 12
The Smurfs games 262
this.container.innerHTML function 162
this.parent() 111
this.username command 171
three-dimensional games
Borderlands 222
Okami 222
Yoshi's Island 222
Zelda: The Wind Waker 222
Tictacti
about 267, 269
example 268
tilesets
about 35
downloading 224
tips, graphics in HTML5 233
tips, sound files 219
trace() function 184
trigger entity, RPG 107
tunes and sound effects
buying 208-210

U

unit testing 159, 160
update() function 83, 91

V

variables, init() method
_choices[i].labelWidth 186
this.cursorLeft 186
this.cursorLeftWidth 186
this.cursorRight 186
this.cursorRightWidth 186
this.Menubackground 187
this.selectedChoice 186
variables, menu init() function
_choices 186
_choice_spacing 186
_entity 186
_fonrt 186
vector graphic design
about 224
drawing 225-229
vegetation_back layer, RPG 76

vegetation_front layer, RPG 77
victory screen
  adding 156-158
virtual buttons 238

W

WAMP server
  installing 7-9
weapons
  adding, to RPG 87
  adding, to side scroller game 129
weapons, RPG
  harm, causing with projectile 91, 92
  projectile, spawning 87-90
weapons, side scroller game
  adding 129
  bomb, building 133-137
  projectile, shooting 129-132
web app 243
web browser
  HTML game, preparing for 236, 237
webhost 29
Webkit-SQLite 174
Weltmeister 24
  collision layer 36
  Is collision layer option 37
  layers menu 36
  layers, rearranging 38
  level1 level 34
  levels, creating 34
  levels, loading 34, 35
  levels, saving 34, 35
  Link with Collision option 37
  Repeat option 37
window.Menu class 185
WordWrap class 109
World of Tanks 262
World of Warcraft 258

X

X 7
XAMPP 7
XAMPP server
  downloading 8
  installing 8
Xcode 12, 27
XDK (Cross platform Development Kit) 25