



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# GameSalad Essentials

Create awesome cross-platform games in no time, and with no coding

**Miguel DeQuadros**

**[PACKT]** open source\*

PUBLISHING

community experience distilled

# GameSalad Essentials

Create awesome cross-platform games in no time, and  
with no coding

**Miguel DeQuadros**



BIRMINGHAM - MUMBAI

# GameSalad Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2015

Production reference: 1240115

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

ISBN 978-1-78439-197-3

[www.packtpub.com](http://www.packtpub.com)

Cover image by Miguel DeQuadros ([miguel@wurdindustries.com](mailto:miguel@wurdindustries.com))

# Credits

**Author**

Miguel DeQuadros

**Project Coordinator**

Harshal Ved

**Reviewers**

Aaron Beaty

Dirk Schembri

**Proofreaders**

Simran Bhogal

Paul Hindle

Clyde Jenkins

**Acquisition Editor**

Neha Nagwekar

**Indexer**

Rekha Nair

**Content Development Editor**

Shubhangi Dhamgaye

**Production Coordinator**

Nilesh R. Mohite

**Technical Editors**

Aakashdeep Kundu

Deepti Tuscano

**Cover Work**

Nilesh R. Mohite

**Copy Editor**

Shivangi Chaturvedi

# About the Author

**Miguel DeQuadros** is a game developer and founder of the independent development studio Wurd Industries, based in Ontario, Canada. He has been developing iPhone games since the release of the App Store back during the exciting release of iOS 2.0. Since then, working under Wurd Industries, he has developed and released nine games and one entertainment app worldwide on the App Store, with more to come.

Originally interested in 3D animation and graphical design, in 2008, he got the game development "bug" and has been developing iPhone apps ever since, which also allows him to use his creativity and knowledge of 3D animation for cutscenes and videos within his apps, and he loves every minute of it. Starting from his first project, *Toy Tennis*, back in 2008 down to his current project, *SpaceRoads*, for the PC, Mac, and other platforms, he now has switched focus away from mobile platforms, and is focusing on PC, Mac, and console-based development. He now primarily uses Unity3D, 3D Studio Max, and the Unreal engine for his current and future projects in an aim to create very high-quality games.

His games can be seen on the App Store on iOS, Steam Greenlight, Amazon, and IndieCity, and of course on his website, [www.wurdindustries.com](http://www.wurdindustries.com). His games have been reviewed on YouTube by Action Soup Studios, who have also interviewed him about the games.

He began expressing interest in GameSalad when it was first launched in March 2009, but never started using it until January 2010 when he started to develop his third iOS game, *iMMUNE 2: Rise of the Salmonella*, a 2D platformer sequel to the iMMUNE series. To this day, he continues as a professional member of the GameSalad community and continues to use GameSalad for prototyping and developing games for iOS devices, and has even branched out to personal computer game development.

# Acknowledgments

I would like to thank my dad, John, even though he called me every 2 seconds to see how my book was going. I'd also like to thank his wife, Lucy, my brother and sister-in-law, Johnny and Katie, my cousin, Corey – even though I haven't seen him in ages- my best friends, Brandon, Kaleb, and Philip, my buddy Marc, at Kfaction Productions for doing awesome music for my games ([www.kfactionproductions.com](http://www.kfactionproductions.com)), Mike and Seanna, ugh! I don't want to leave anyone out! Let's not forget a very special thanks to my wonderful wife, Joanne, for encouraging me during the writing of this book (my wife told me to thank Stella and Mimi as well... Our dog and cat, as if Stella didn't annoy me during the writing stage). Writing a book is very difficult, especially on your spouse and friends, as it does limit your association. I would also like to thank Packt Publishing and all the wonderful employees who helped me out, both in the initial stages and into the final chapters of the book; your professionalism and ability to clearly explain things also helped me a lot. Thank you Neha, Dayan, and Shubhangi for everything; you made the production of the book a very smooth and enjoyable process!

# About the Reviewer

**Aaron Beaty** is a recent graduate from Bradley University with a degree in interactive media, specializing in video game design. He spent his college career learning and mastering the art of developing video games and learning the ins and outs of the design process. Recently, he independently developed a game that has made it onto the iOS and Android markets. The game is titled, *The Interruption System*, and is currently available for free download.

In his spare time, Aaron devotes his time to continuing to expand upon his development background by researching the latest news and development tips in gaming. He also spends a lot of time writing poetry. He utilizes his poetic skills to assist him in building stories for games and expanding upon games that he's already played. Other than designing games and writing, he plays an extensive amount so that he can continue to learn from what other developers have done. What's a game designer if he doesn't know the game he's attempting to design, right?

This is his first time editing a book for use in the public. He was a proud contributor to this book, and he hopes that in the future, he will be able to contribute to future publications. He also hopes that you enjoy learning the beginnings of game development and continue to utilize the skills this book presents for future use, whether it is hobby or education.

---

I would like to thank my mom, dad, and brother for helping me throughout my entire college career and supporting my decision to become a game designer. I would also like to thank the Lord for keeping my feet planted firmly on the ground and for helping guide me towards my future.

---

[www.PacktPub.com](http://www.PacktPub.com)

## **Support files, eBooks, discount offers and more**

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## **Why Subscribe?**

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## **Free Access for Packt account holders**

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Setting Up GameSalad</b>	<b>5</b>
<b>Looking into version 0.11</b>	<b>5</b>
Downloading and installing GameSalad	7
Why do I need an account?	9
A quick tour of the interface	10
Tabs in GameSalad	13
<b>Summary</b>	<b>18</b>
<b>Chapter 2: From Design to Production</b>	<b>19</b>
<b>The design process</b>	<b>20</b>
Storyboards	21
<b>From paper to screen</b>	<b>22</b>
<b>Production</b>	<b>29</b>
Creating our Project	29
<b>Summary</b>	<b>36</b>
<b>Chapter 3: Diving In – Introductions and Menus</b>	<b>37</b>
<b>Logo animation</b>	<b>37</b>
<b>Creating the main menu</b>	<b>44</b>
Selecting levels	47
<b>Creating our introduction</b>	<b>49</b>
<b>Working with particles</b>	<b>56</b>
<b>Adding emotion to the scene</b>	<b>57</b>
<b>Summary</b>	<b>58</b>

*Table of Contents*

---

<b>Chapter 4: Diving In – Player Creation</b>	<b>59</b>
<b>Controls and player movement</b>	<b>59</b>
Mobile controls	65
Attacking	70
Melee weapons	74
<b>Summary</b>	<b>76</b>
<b>Chapter 5: Diving In – Gameplay Mechanics</b>	<b>77</b>
<b>Health, dying, and lives</b>	<b>78</b>
<b>Scoring and Saving Games</b>	<b>81</b>
<b>Power ups, ammo, and inventory</b>	<b>86</b>
<b>Ammo</b>	<b>89</b>
<b>Summary</b>	<b>98</b>
<b>Chapter 6: Diving In – Enemies</b>	<b>99</b>
<b>Creating enemies</b>	<b>99</b>
Motion and path finding	101
Attacking	104
Enemy health	110
<b>Summary</b>	<b>112</b>
<b>Chapter 7: Diving In – Objectives</b>	<b>113</b>
<b>Objectives</b>	<b>113</b>
<b>Way finding</b>	<b>119</b>
<b>Boss battles!</b>	<b>120</b>
<b>Multi-platform publishing</b>	<b>124</b>
<b>Spreading the word about your awesome game!</b>	<b>127</b>
<b>Summary</b>	<b>128</b>
<b>Appendix: The Challenges of Game Development</b>	<b>129</b>
<b>Examples of developers who made it big</b>	<b>130</b>
Phil Fish and FEZ	130
Edmund and Tommy-Super Meat Boy	131
<b>The challenges of game development</b>	<b>133</b>
<b>How GameSalad makes it all easy</b>	<b>134</b>
<b>Index</b>	<b>137</b>

---

# Preface

Have you always wanted to learn how to create an awesome game, but never had the time to go and learn a new programming language? Then this book is for you! We will harness the power of GameSalad, one of the greatest game development platforms out there. You will learn how to create a complete game through the course of the book. Best of all, you won't have to write a single line of code! GameSalad is a behavior-based development system with a drag-and-drop interface.

## What this book covers

*Chapter 1, Setting Up GameSalad*, teaches you how to sign up for an account for GameSalad, download the software, and install it on your system. We will then go through the interface to see what does what.

*Chapter 2, From Design to Production*, covers the process of designing your game, from a game design document to level and character design. From there, you will learn how to take your sketches from the design document and put them on the screen using imaging software to create "sprites". Then, you will start producing your game.

*Chapter 3, Diving In – Introductions and Menus*, teaches you how to create a logo animation for your game, as well as introductions and a main menu. We will make full use of GameSalad's tables feature as well.

*Chapter 4, Diving In – Player Creation*, teaches you how to create the main character from the images we created earlier. We will discuss player movement, controls, collisions, and more!

*Chapter 5, Diving In – Gameplay Mechanics*, teaches you how to create health for our player, a scoring system, lives, and an inventory system.

*Chapter 6, Diving In – Enemies*, teaches you how to create enemies, which will include movement with a great path finding system, health, attacking, and more.

*Chapter 7, Diving In – Objectives*, teaches you how to integrate level objectives into your game. We will discuss a way finding system, boss fights, and finally we will discuss publishing your game and how to make money off it!

*Appendix, The Challenges of Game Development*, covers the hardship of independent game developers along with the journey of two of the biggest indie titles ever released – *FEZ* and *Super Meat Boy*.

## What you need for this book

A Mac or a PC, your favorite imaging software, such as Photoshop or Gimp, and GameSalad!

## Who this book is for

This book is for all those out in the world who have always wanted to create their own game but have never been able to learn, or been able to take the time to learn, a whole programming language. With GameSalad, you can create a complete multi-platform game in half an hour. If it's always been your dream to create a game but you have no idea where to start, this book is for you.

## Conventions

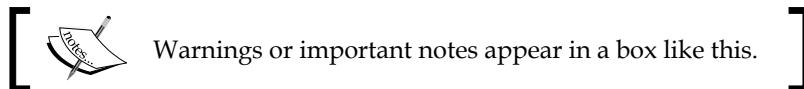
In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
CGRect playerRect = [player collisionBoundingBox] ;  
CGPoint playerCoord = [layer  
coordForPoint:player.desiredPosition] ;
```

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: "If this is your first time uploading your project, click on **Publish as New Game**."



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](mailto: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](http://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.

## Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: [https://www.packtpub.com/sites/default/files/downloads/B03956\\_1973OS\\_Graphics.pdf](https://www.packtpub.com/sites/default/files/downloads/B03956_1973OS_Graphics.pdf).

## 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.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

## 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](mailto: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](mailto: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

## Setting Up GameSalad

Do you have an amazing idea for a game? Do you want to create the next **Super Mario, Portal, or Half-Life?** (The gaming community could probably release the third instalment of Half-Life sooner than Valve, but hey! I joke.)

But *wait!* I know what you're thinking, "Man! I have no idea how to do any of that fancy fandangled coding." That's why you've come to the right place! I'm going to guide you through a fantastic program called GameSalad. What is GameSalad you ask? It's a great studio for creating multi-platform, super powerful games, and requires no coding! Everything is behavior based, but we'll get into that a little later. In this chapter, we are going to look at some of the following:

- What's new in GameSalad version 0.11
- Downloading and installing GameSalad
- A quick tour of GameSalad's interface

For those of you who have used GameSalad in the past, this is just a refresher course; for those of you who haven't, sit tight and enjoy the ride! First, let's see what's new in GameSalad.

### Looking into version 0.11

There have been a lot of changes in GameSalad recently. From a nice redesign of the interface to many new features and changes; it's worth looking into.

So, what exactly is new in version 0.11?

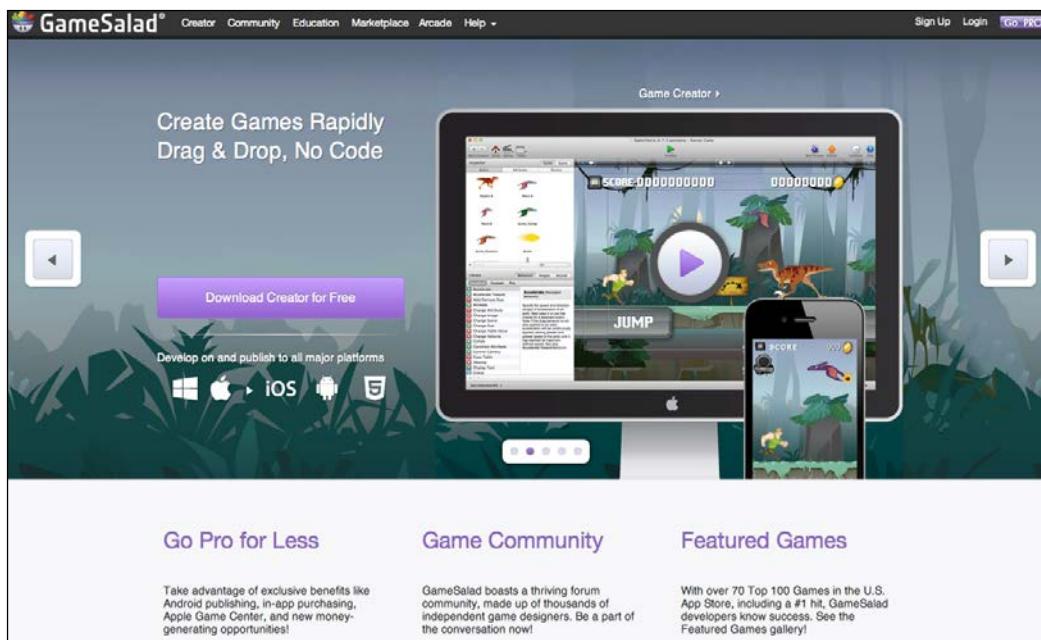
- **Massive improvements to the engine:**
  - The website announced over 240 bug fixes. 240! That's a lot of bugs to squash, but the GameSalad developers have done a great job.
- **Network behaviors:** You can now sync information from any server. This really opens up a lot of game development possibilities! Now you can store data remotely, and you can even create a turn-based game all synced through servers! Awesome! Available on a pro account only.
- **New monetization options:** New ways to make money! **Chartboost** and **RevMob Ad Networks** have been added to the software, which means more ads being inserted into your game, equalling more profit for your game.
- **Looping behaviors:** I'm super excited about this one! Looping and loop over table behaviors reduce the complexity of your project by far! For example, what would have taken 10 behaviors now will only take 1 or 2 at the most.
- **Enhancements to text and strings:** You can now display text relative to the position of an actor. This alone is exciting, but there's more. New display options, substrings, and replace and find functions allow complex string editing.
- **Optimized rendering:** The whole rendering system for images has been revamped, which drastically improves the performance of your game. I've seen this already running some of my old projects in version 0.11
- **New multi-touch options:** Now you can detect up to 11 touches at a time, increasing the complexity of gameplay that you can deliver to the world.
- **Custom loading icon (pro accounts only):** Instead of the standard loading icon, you can now create your own. Thus, this allows you to unify your design throughout the game.
- **And more features:** Tables (a super powerful system we will use throughout this book), changes to the **Change Scene** behavior, **Google Play Tablet Support** (for pro users), stretch mode allowing you to develop for the iPhone and deploy to a PC (GameSalad will compensate for the screen size difference and stretch the screen accordingly), tweaks to the rule system, and changes to the **OpenURL** behaviors.

All these changes are drastic, and you will see how great it is when you dive into GameSalad. If you've used GameSalad in the past, you are going to notice a massive difference in performance and ease of development. I'm already astonished at the bug fixes. Oh! How many projects I've lost to crashes and bugs! Now we don't have to worry about any of that!

Now that you know what's new, let's get it on our computers.

## Downloading and installing GameSalad

Let's head over to [www.gamesalad.com](http://www.gamesalad.com); while the creator is downloading, let's register an account as well.



## *Setting Up GameSalad*

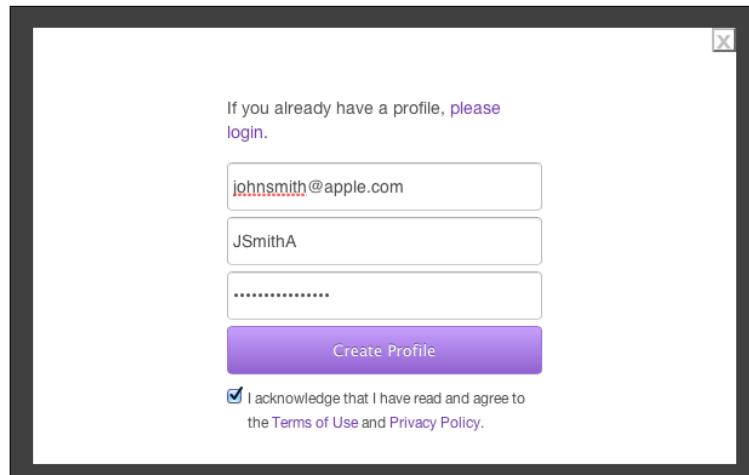
Right on the front page, there's a button to download GameSalad Creator. Click on that button and you will now be greeted by a screen requesting your e-mail address. Fill yours in and check the **I am over 13 years of age and have read and agreed to the Terms of Service** checkbox. GameSalad will start downloading. You are now one step closer to creating your very own awesome game!



The current version, at the time of writing this book is 0.11 for Mac and 0.10.5 for Windows.



Now, while you're waiting for the download to finish, let's create an account. On the top-right corner of the GameSalad banner, you will see an avatar button next to a drop-down arrow. Click on it and it says **Sign Up**; click that bad boy. There will be a pop-up box asking for your information. Simply fill it in. This is shown in the following screenshot:



After this, you will receive an e-mail asking you to activate your account. Once you do, boom! You're done! That was super easy.

## Why do I need an account?

Good question! An account is required for publishing, but you also get great benefits such as:

- Access to the GameSalad forums
- The ability to view your portfolio (all the games you've uploaded are stored on the site so you can edit them at any time)
- Purchase graphics, audio, and templates from the GameSalad Marketplace for use in your game
- Go Pro!

Now, let's install GameSalad. This is pretty standard stuff, but if you don't know how, let me walk you through it.

### Mac installation

Locate the GameSalad installation file (normally downloaded in your Downloads folder) and open it. Agree to all EULAs (End User Licence Agreements). The installation will now show up. If it doesn't, go to your desktop and locate the GameSalad Creator DMG that was just mounted and simply click-and-drag the **GameSalad** icon into the **Applications** shortcut, as shown in the following screenshot:



GameSalad installation for Mac

Once this is done, you will have officially installed GameSalad on your Mac!

## Windows installation

Locate the saved installation file and open it. You will be prompted throughout the installation process. Once it's done, all you have to do is open it.

[  Some users have reported issues with the installation stating they need to check their Internet connection. GameSalad installation requires an active Internet connection. Some users have reported simply re-downloading the install file works, while others report using an alternate Internet browser such as Internet Explorer 9, Mozilla Firefox, Google Chrome, or Safari works as well. All you need to do is drag and drop your installation file on one of these browsers. ]

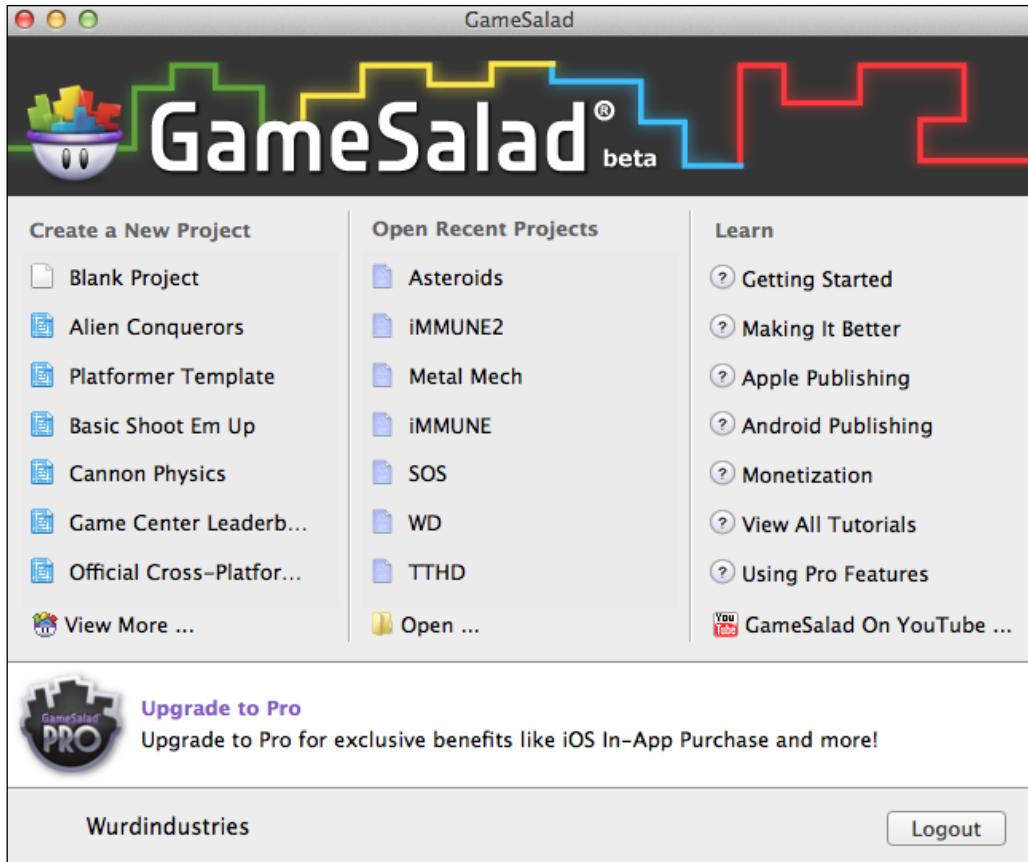


Easy peasy lemon squeezy! Once that's done, let's dive in to GameSalad for a quick overview of the new interface.

## A quick tour of the interface

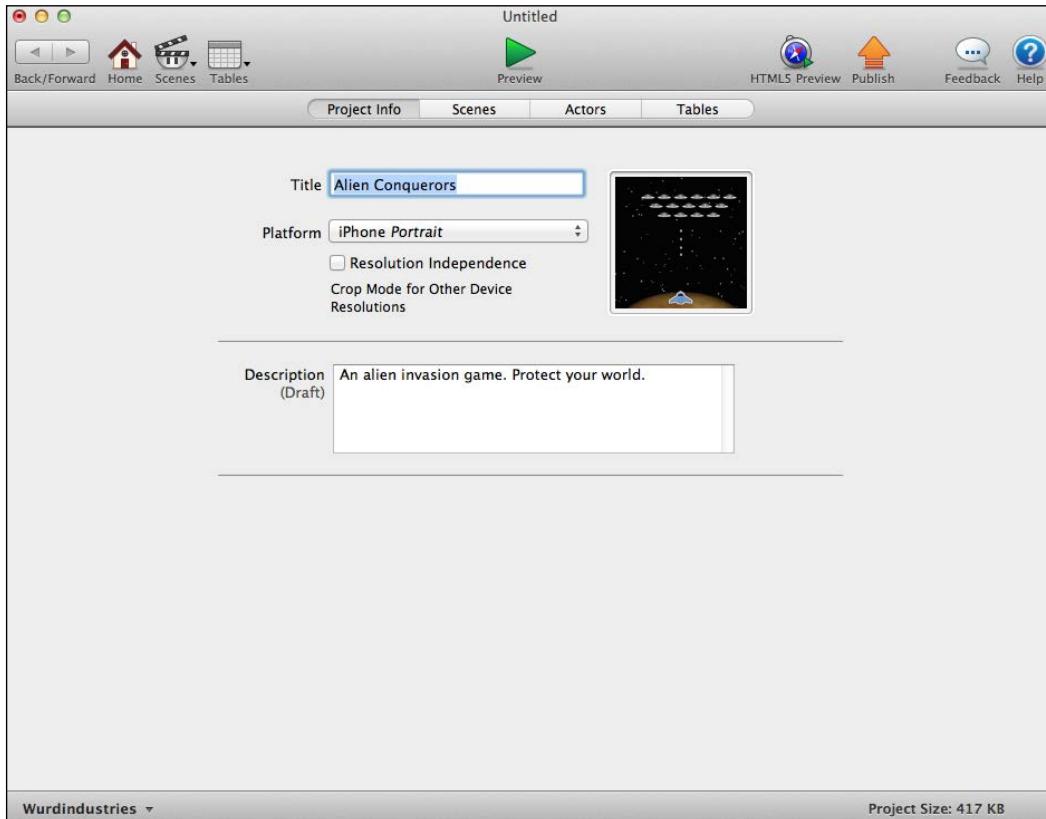
Now that we have installed GameSalad, let's open it. For Mac, go to your Applications folder and open **GameSalad**. For Windows, the default location is under Program Files | GameSalad | GameSalad Creator. Or you can find it in the Start menu.

This is the screen you will be greeted with if you are running GameSalad on Mac OS X; the new main menu is as shown in the following screenshot:



If you are running Windows, you see a brand new blank project. From here you can open up tutorials, your recent projects, or create a new project either from a blank project or through template projects. If you are running Mac, for now, let's open up the **Alien Conquerors** template project. If you are running Windows, you can download template projects from the GameSalad website under **Help | Cookbook Tutorials | 6. Windows Creator | 6.7 - Windows Creator Templates**. You will then find all the templates. Click on the **Alien Conquerors** template to download the project.

Now we are met with the **Project Info** tab, as shown in the following screenshot:



On this screen, you can set the title of your project, whatever platform you want to develop for, and a description. We will start with what the buttons do:

- **Back/Forward** buttons: These buttons allow navigation through the screens we will see throughout our development.
- **Home** button: A fast way to pop back to the **Project Info** screen.
- **Scenes** drop-down button: A quick way to jump to scenes without having to go back to **Home** and select the **Scenes** tab.
- **Tables** button: Allows you quick access to your created tables.
- **Preview** button: Plays your project. If you have a scene opened, it will play that scene; but if you are at the home screen, it will play the whole project.
- **HTML 5 Preview** button: Simulates your project in HTML5.

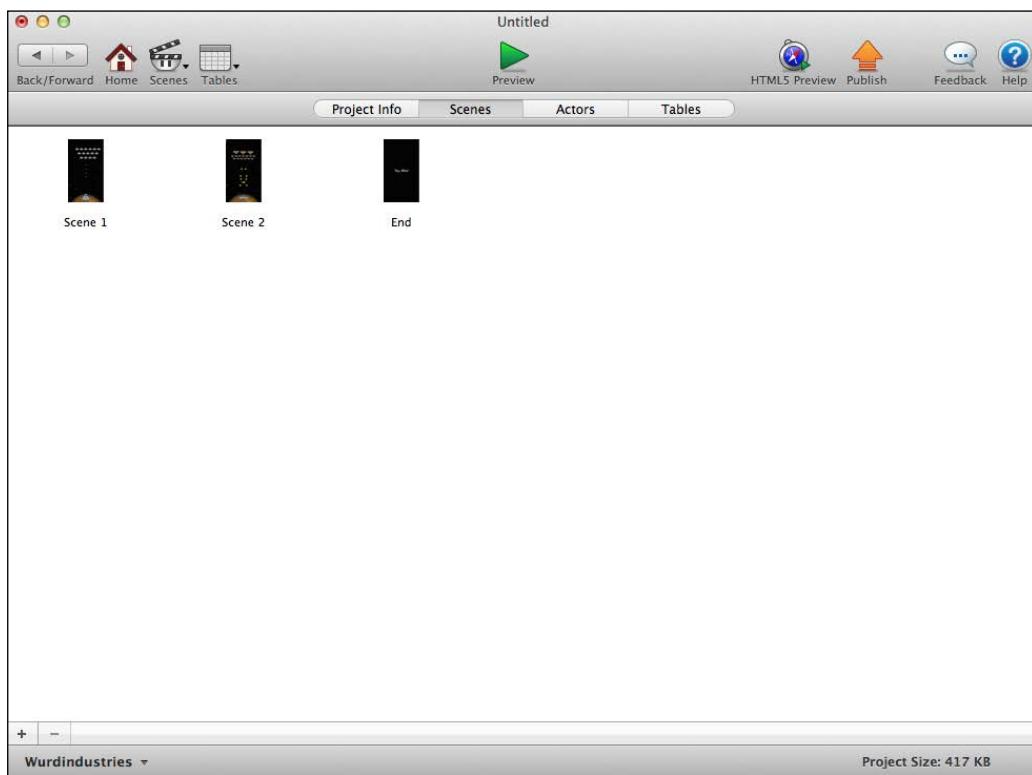
- **Publish** button: Allows you to publish your project. (We will get into this later.)
- **Feedback** button: Opens up your default Internet browser and allows you to send feedback to the GameSalad creators.
- **Help** button: Opens up your Internet browser again and brings you to the GameSalad cookbook, a quick reference for all things GameSalad.

## Tabs in GameSalad

Tabs in GameSalad are placed so that we can navigate through the interface. Let's take a close look at what the various tabs do in the interface. Let's first discuss the **Scenes** tab.

### Scenes

Click on this tab and you will see the different levels you have created. You can see in this template project that there are three scenes – two levels and one for a "You Win" screen.



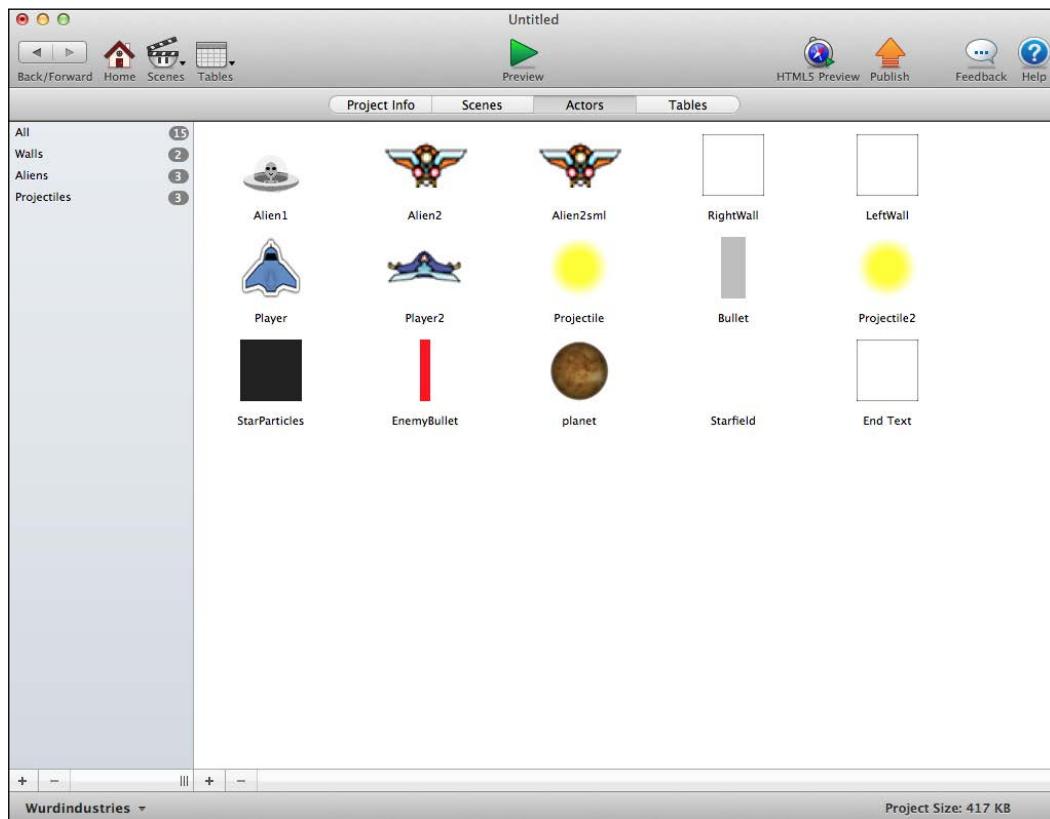
## *Setting Up GameSalad*

---

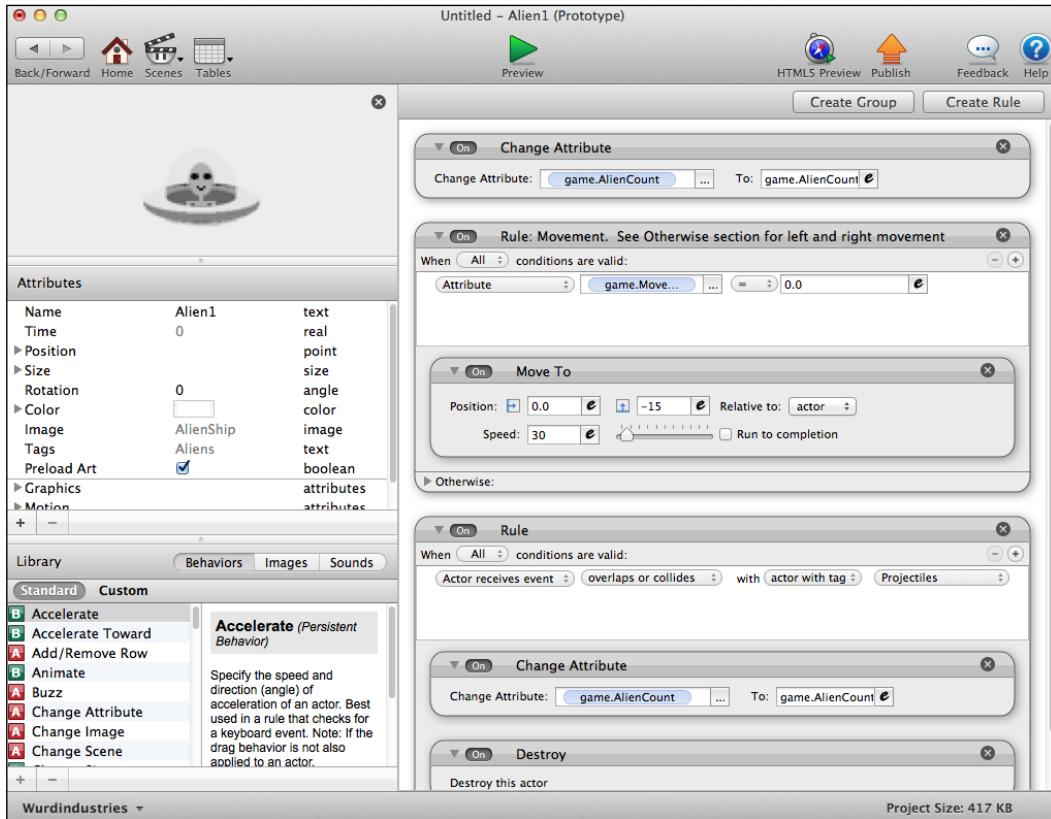
At the bottom of this screen, you have a '+' and a '-' button, which allow you to create (+) and delete (-) scenes. Give it a try. Click on some of the scenes and explore the levels a little bit to see how things work. Again, we will get into more details on this later.

## Actors

This tab shows you all the actors and objects you have created throughout your project. What are actors? Actors are the visual objects in your game. Actors can be characters you play, enemies, obstacles, platforms, background objects, and so on. Here you can create and delete actors as well as manage them. You can also create tags and arrange the actors accordingly. These are all things we will discuss later. Look at all the potential you have here. Can you picture the awesomeness you are about to create?



When you start editing these actors, you will be greeted with the actor editor, as seen in the following screenshot:

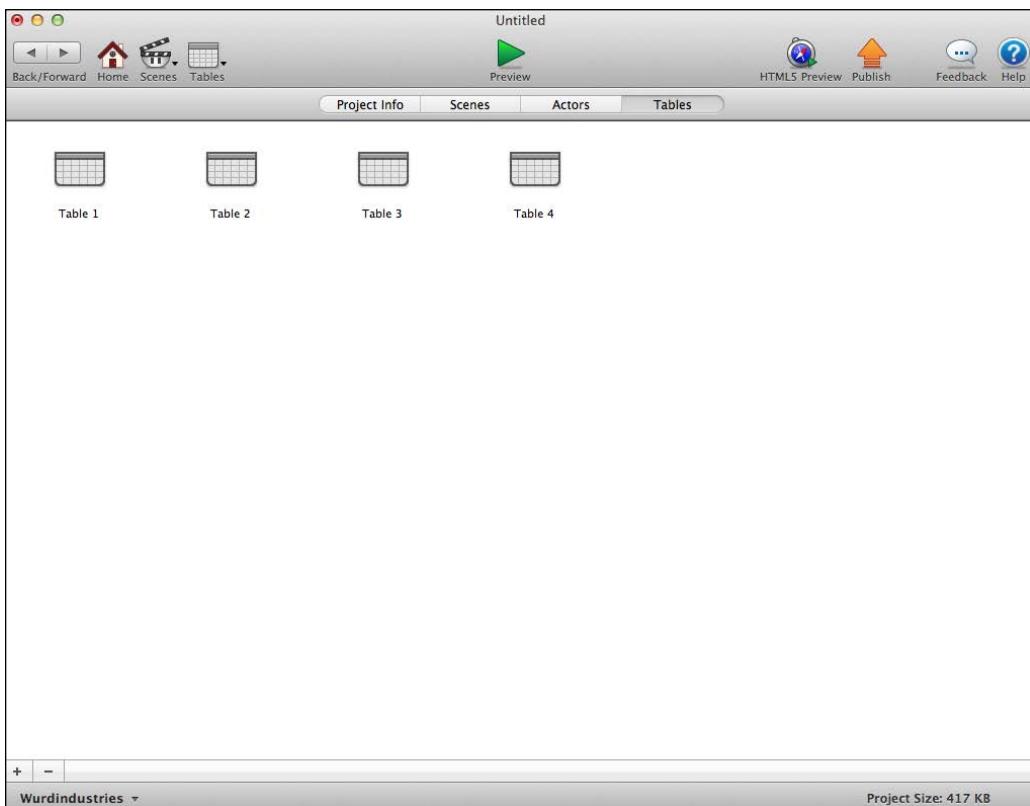


The actor screen

Scroll through some of the behaviors or actions and get an idea of the things you can do. I'm sure you'll see you can get into some pretty powerful rules and behaviors.

## Tables

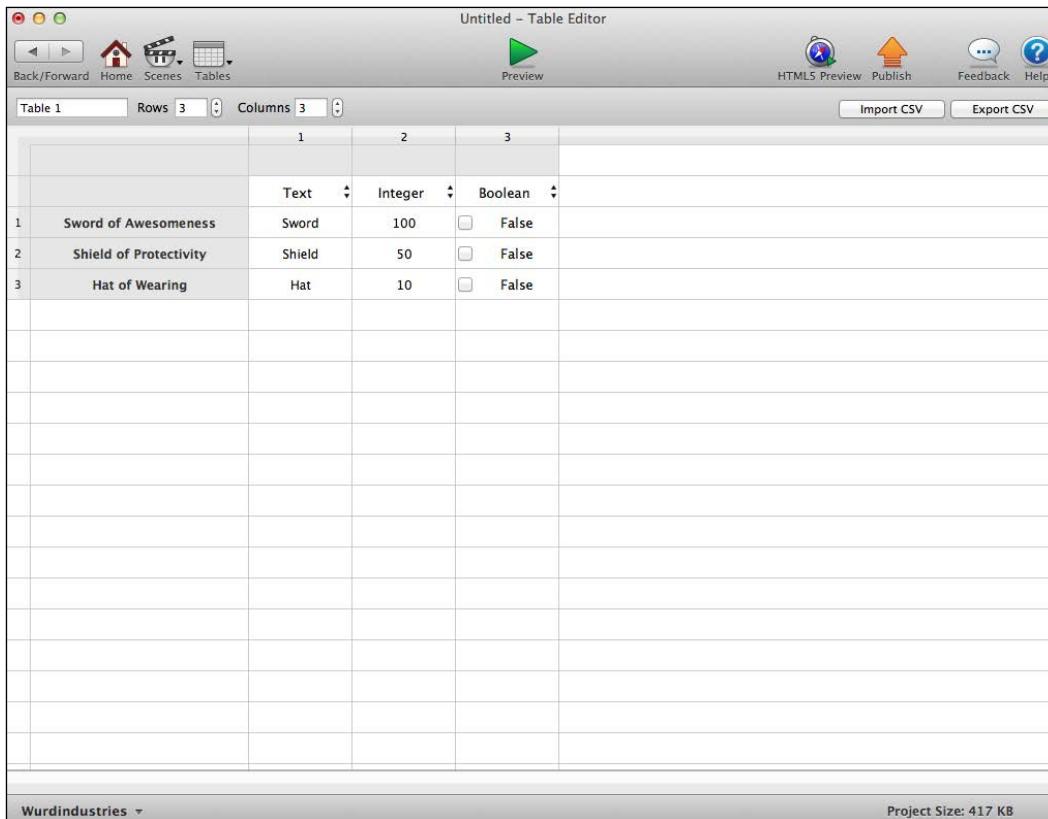
From here, you manage all your tables. What are tables? Essentially, they are a great way to store data and information. Think of it as an Excel file that stores all the players' items, scores, and information. GameSalad has the ability to save information into them and load them all on the fly. This project we opened didn't have any tables created, so I added a few just so you can get an idea of what this screen looks like when populated. When you click on a table, you will essentially open an Excel file. You can add rows and columns and populate them accordingly. You can also import and export CSV files, if you prefer to use your own editor. We will get into using tables later in this book.



I did some test populating of a table. These would be items available within your game. As you can see, I created a few items just so you can see how the table editor works.

The Text column is obviously the name of the item. The Integer column is the value of the item, should you choose to sell it in-game. The Boolean column is to detect whether you already own the item in your inventory.

*Boolean*?! What's this trickery? A Boolean is a complicated way of saying whether something is true or false. Let's check out the table in the following screenshot:



## What your tables look like when populated

Do things seem easy to navigate? They sure are. Don't worry if you don't understand some interface elements right now. We are going to get into them in the next few chapters.

## **Summary**

In this chapter, we learned about the great new features in GameSalad 0.11, and we downloaded GameSalad and set up our own user account. We also had a quick overview of the GameSalad interface. This is all preparation for using the software, getting to know what's new, and the advantages of having an account. An interface tour will help you in your game development.

As we continue through this book, we are going to get as deep into the interface and development as we can get; but don't worry, it's not hard at all! Hopefully, by the end of this book, you will be able to take what you've learned and create your own awesome game!

In the next chapter, we are going to discuss the design process-how to design your game and get things into production.

Sit tight... It's going to be a fast-paced ride!

# 2

## From Design to Production

All great selling video games start off with a design phase. Think of fantastic games like **Portal 2**, **Destiny**, or **The Last of Us**. All these games had a huge production and design phase. It would be a little unrealistic to think such amazing games started with the director saying, "Hey guys! Start creating some levels, we'll figure things out later." Granted, sometimes it is necessary to create games like this to get your creative juices flowing, but for the most part, it's a great idea to design your whole game first, and then start developing. This will certainly streamline the development because you wouldn't have to think as you go. That's what we are going to discuss in this chapter; how we can go from the design phase into production.

Let's see what we are going to cover:

- Designing your game
- Creating **sprites** for your game
- Starting the production of your game

Now, the fun really begins. From this chapter on, we are going to create your first game (for some, anyway).

First, let's talk about the design process.

## The design process

Already have an awesome idea for a game? Great! Now put it down in writing and try to be as detailed as possible. You may have heard of a design document, which is exactly what it sounds like; a document full of details of your game, whether it's story content, weapons, items, characters, and so on. Here's a good baseline template for your design document:

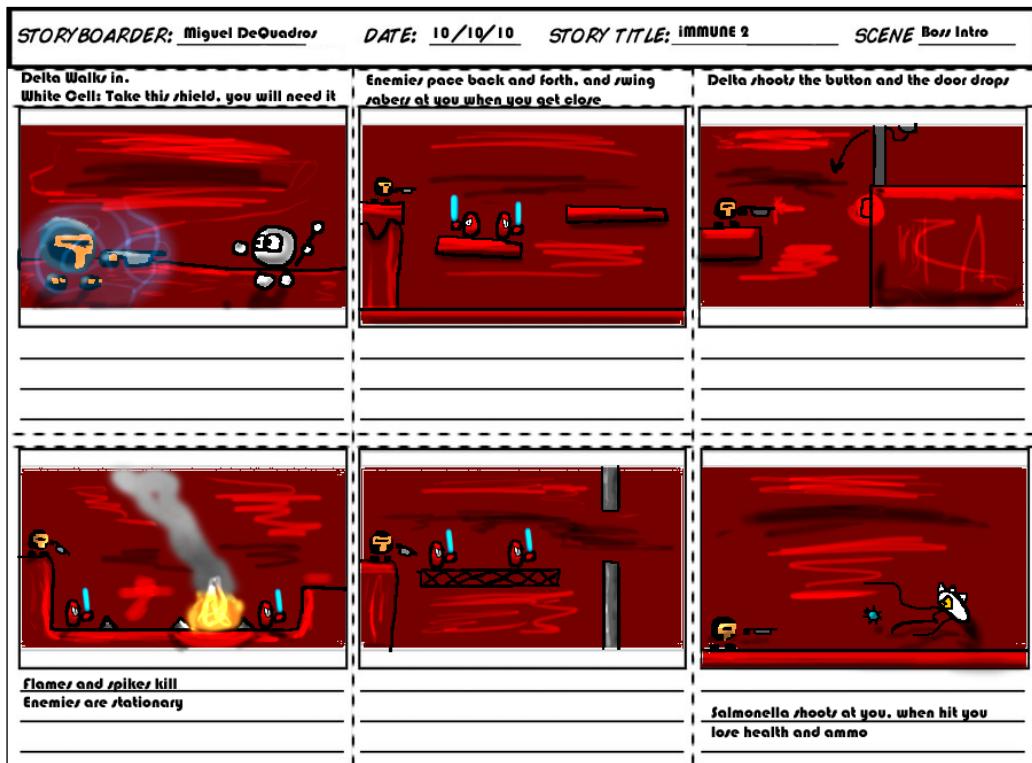
- Name of the game
  - Storyline
  - **Overall environment of the game:** Do you want it to be comedic, or dark?
  - **Objectives:** Things your player will have to do to resolve the issues in the game
  - **Game Progression:** How is the game going to play out? Slow and steady, or fast?
  - **Features of the game:** We are talking about SDK (Software Development Kit) level here. What do you want to include? **Game Center, Xbox Live, Leaderboards**, or online multiplayer?
  - **Characters:** This is a fun one to work on. Who is the main character? What's his (or her) background story? What does he (or she) look like? What are they wearing? Who is the girl you meet in the next town? Who is the antagonist? There are so many details you can go into here. Have fun!
  - **Controls:** How are you going to control your player? Touch screen, keyboard, or mouse? What does each button do?
  - **Level design:** This is an interesting one to design too. For example, you start off in your home. What does it look like? What objects can you interact with?
- Now you can go ahead and actually draw out the level.

This is a much stripped down version of a game design document. You can go into a ton of details, and remember, the more details you include, the better. There are many elements to the game design phase. The next element of the design phase is storyboards for planning out cutscenes.

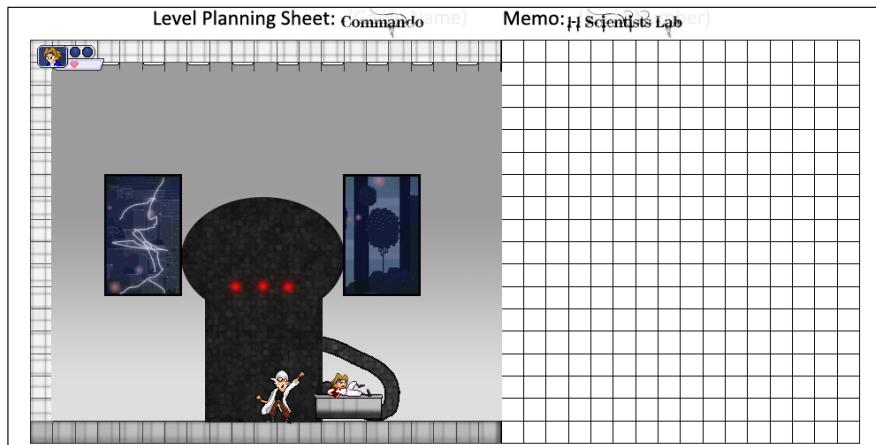
## Storyboards

Another part of the design phase would be storyboards. These are critical for planning out cut scenes, but can be used to show level progressions as well. The screenshot in the following is a storyboard I created for one of my games back in 2010. This was for one of the final levels. It starts off with a quick **cutscene**, then moves on to each stage of the level. At the top of the storyboard, you have all the details of the page, such as the date it was created on, the name of the level or game you're working on, and the scene.

Above each board, you have a description, and below you can write comments such as, "Shoot padlock on door to open, door flies open to reveal hordes of zombies."



Also, here's an example of a level design document. (This is for 2D games only; 3D games would need more perspective drawing):



As you can see in the above figure, in the **Level Planning Sheet**, you want to add in all the details of the level. Granted, this doesn't have to be as detailed as this image, but it's a good idea to be as detailed as possible. Things like colors and objects within the level can be tweaked later on to add some more detail. Creation of these level planning sheets can be done on graph paper, or you can even create them using your favorite imaging software, such as MS Paint or Adobe Photoshop.

Level planning sheets and storyboards are a critical key to the game design phase. They can add as much detail as possible. So, when you start developing the game, it cuts down the production time considerably. Personally, I have a binder full of blank-level planning sheets and blank storyboard pages for my game design, as well as a design document template like the one I showed you earlier. You can print out blank planning sheets and storyboard pages in the resources section for this book.

Planning a game can take hours, days, or months, but when you actually dive into the design process, there's a lot of fun that can be had. Once your creative juices start flowing, you will get on a roll. Feeling stumped at a certain point? Don't worry! You can move on to something else, then go back and try again. It's always a good idea to take a break so you don't burn yourself out.

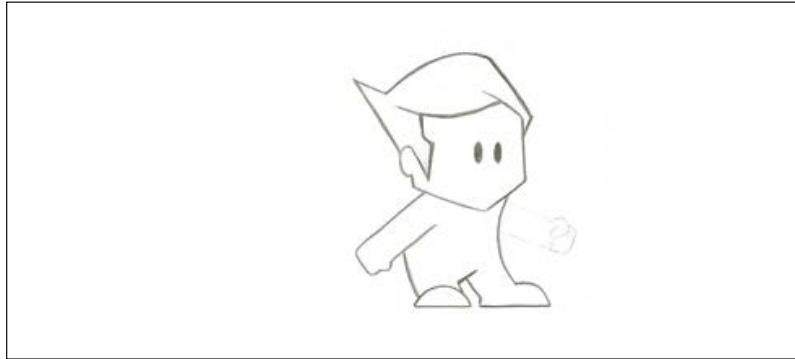
Now that you have your game all planned out, let's put our ideas on screen!

## From paper to screen

Make sure your whole game is designed, and carefully planned with every detail. What now? Well, let's take it from your paper planning and create sprites for the game.

What are sprites? Sprites are the images that make up your game, from your player to items and environment elements; each image you create is a sprite.

Drawing your sprites are really not difficult, in fact you can even scan your drawings into Photoshop and draw on top of your paper drawing. Incidentally, we are going to do something similar. Here's a drawing of a simple character:



Credit: williammalone.com

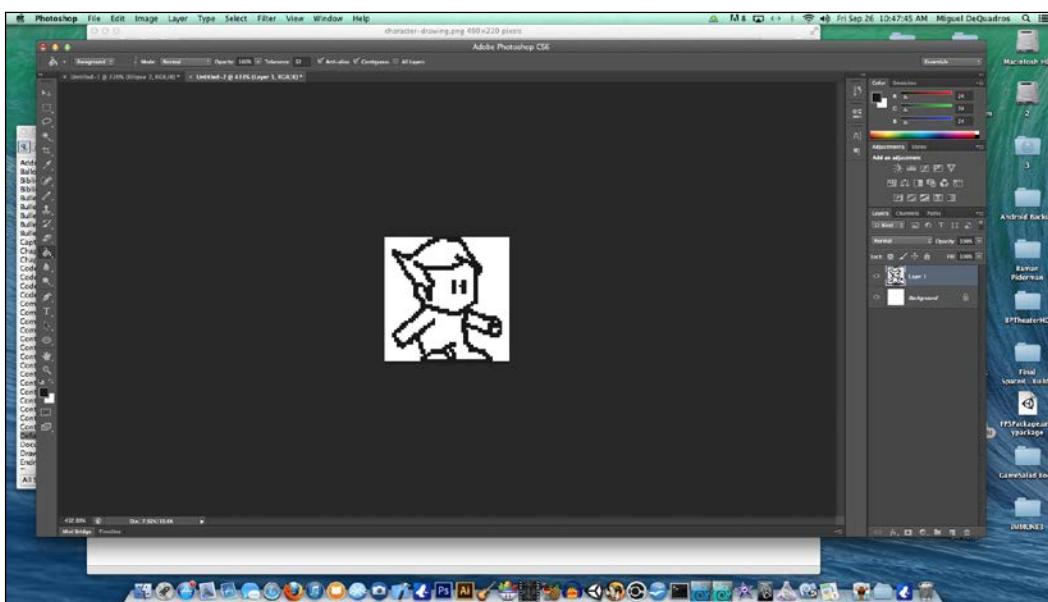
Nothing fancy, but certainly something simple we can work with. Let's open up our scanned image in Photoshop.

Typical sizes for sprites are 32 x 32; however, with bigger screens such as the **Galaxy Note** and **iPhone 6 Plus**, we will want to use some larger sprite sizes. For this example, we are going to create a 50 x 50 sprite. We will worry about sizing it later.



Once Photoshop is opened, you will want to create a new layer on top of your image so you can simply delete your drawing layer once you are done tracing.

All you have to do is use the pencil tool to trace out your character. The pencil lines don't have to be exact, and you can even add details to the character you hadn't originally drawn. If you want to make more exact images you can consider purchasing a drawing tablet for your computer; Wacom is an excellent brand and very accurate for drawing on your computer. If you do not want to purchase one, don't worry! Tracing with the mouse can be just as accurate if you take your time when tracing, and stop when you have to draw a new line. This way, if you make a mistake on the current line you're drawing and click on **undo**, you won't lose a whole piece of your drawing. After tracing I copied and pasted it over to a 52 x 52 blank document, and scaled down the tracing.



You may need to do some adjustments to your lines when you scale it down. I just used the **paint bucket tool** and filled in the lines.

For coloring everything in, we are just going to use solid colors right now. We will worry about the shading later.



Once everything is colored in, we will want to do the lighting, that is, shadows and highlights. If you need to figure out the matching colors, use the **eye drop tool** to pick a color, then open up the color picker and simply select a darker or lighter color with the same hue.



The eye drop tool

The following picture shows the image we see after we use the eye drop tool:



Looks much better! It adds a lot of detail to the character. From here, you can go on and add more details, such as buttons on his shirt, facial expressions, or logos on his shirt.

Now, let's talk about animating the character. For almost every object you create within your game, you will need to create animations for them. This can be an idle animation that plays when the character is sitting doing nothing, or walking, jumping, and running. Animating your character is relatively simple. Take a look at the next few images. This is a quick and easy walking animation that is composed of three images: one with the left foot forward and the right foot back, both feet centered, and right foot forward with the left foot back.

You can even use one of the walking images for a jumping image.



The first image of the walking animation is the character with his left foot forward and right foot in the air. Think of how you walk, or watch videos of people walking to get an idea of how to draw these animations.



Next is both feet aligned, almost like a standing animation. After the previous image, the right foot is coming down and the left foot is moving back.



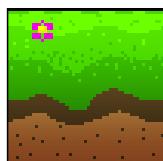
Finally, the right foot is forward and the left foot is up in the air. There's no need to create another image with both feet aligned, as we will reuse the previous image.

These are very simple animations, and once you get into actually animating them in GameSalad, as we will see later in this book, if you find they don't look right, you can always tweak them later.

The same principals apply to designing weapons and items.



Also, platform tiles and environment elements follow the same rules.



Speaking of environments, I find good backgrounds and foregrounds can make or break a game. What I like to do is add multiple layers of background and foreground elements within GameSalad. This gives it a parallax background look, and these can be animated as well.

What are **parallax backgrounds**? Parallax scrolling is a special scrolling technique in computer graphics wherein background images move by the camera slower than the foreground images, creating an illusion of depth in a 2D video game and adding to the immersion.

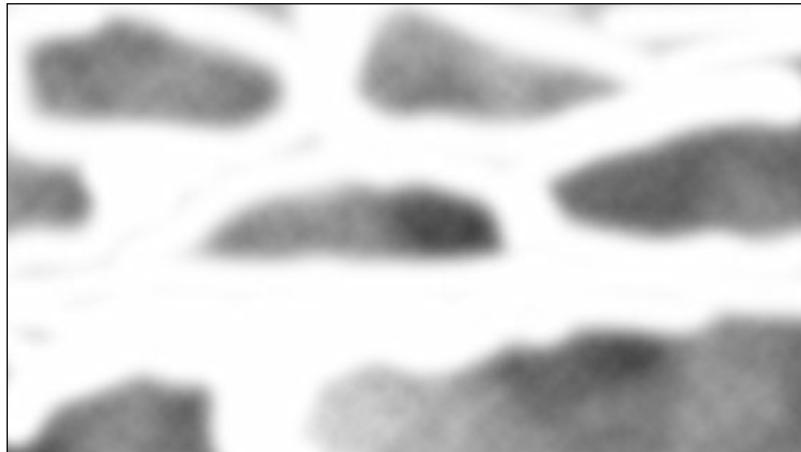
For this example, I created three different layers that will compose the background: one is the sky, the middle is a city scape, and finally, a cloud layer. Here's what they look like separated:



The furthest background will be the sky:



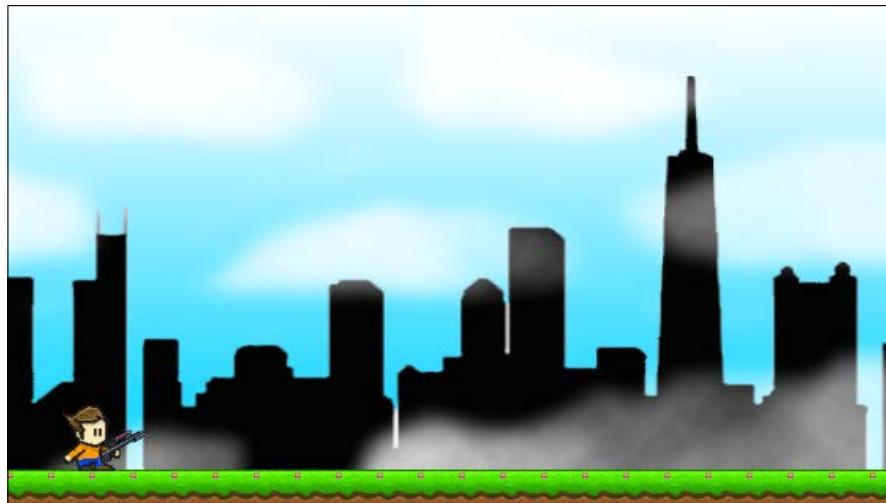
A nice city scape as the backdrop element:



Now, having composed all three images together in GameSalad with the preceding cloud layer having an additive blending mode, everything looks awesome! We will later see how we can do some cool atmospheric effects such as fog and bright lights.



When creating background images, it's always a good idea to make them with **tiling** in mind. What's tiling? It's a wrapping method in GameSalad that will repeat the image an infinite amount of times when we stretch the actor. If you utilize tiling to make the image tileable, the image with it will repeat seamlessly. (The images I created are not tileable.)



See how much fun this is? I love game development! Hopefully, you will too, by the end of this book.

Now, it's time to start using GameSalad!

[  Tip: It's always a good idea to have a specific folder on your computer for all your game assets. It saves a lot of time when importing your images or sounds because you don't have to go searching; everything is right there in one convenient spot. ]

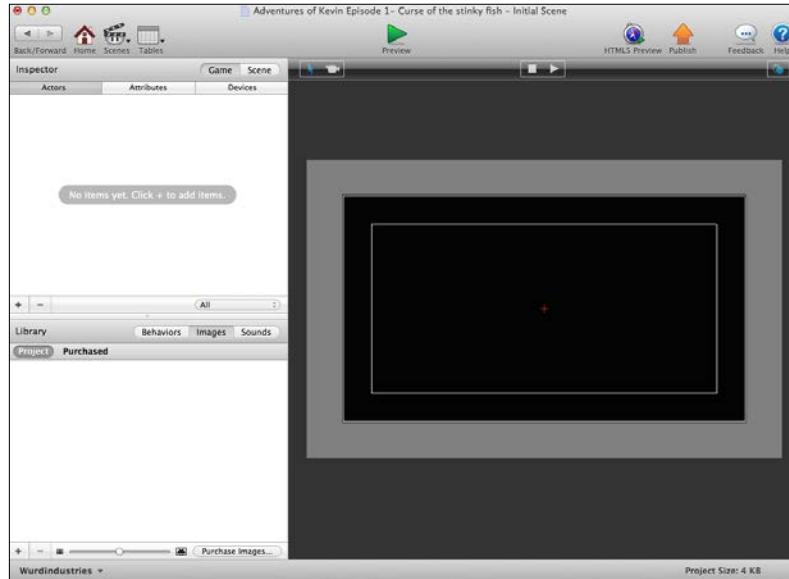
## Production

From the images you created, or even ones you may have downloaded, we are going to start using them in GameSalad. We are going to see how to import images, make actors out of them, and start laying out some levels.

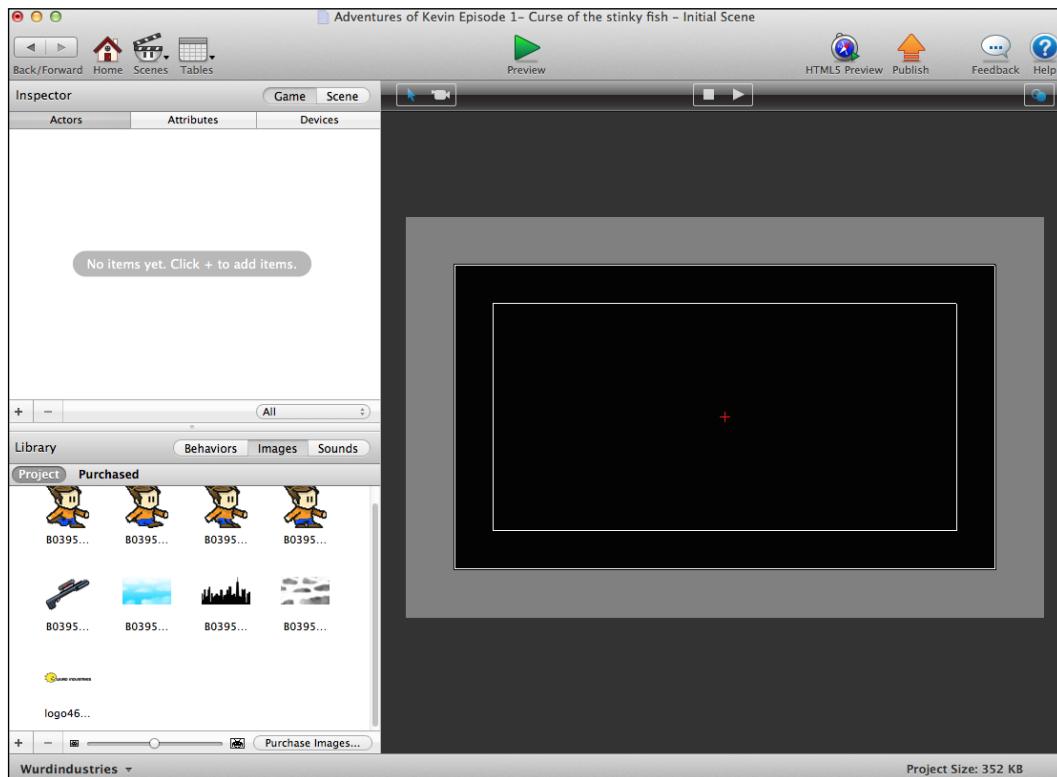
### Creating our Project

Let's start with our project.

Open up GameSalad, and under **Create a New Project**, click on the **Blank Project** option. You can fill in all the details of your game if you like. I'm going to call the project title "**Adventures of Kevin, Episode 1: Curse of the Stinky Fish**". Yep! Click on the **Scenes** tab and open up the **Initial Scene**:

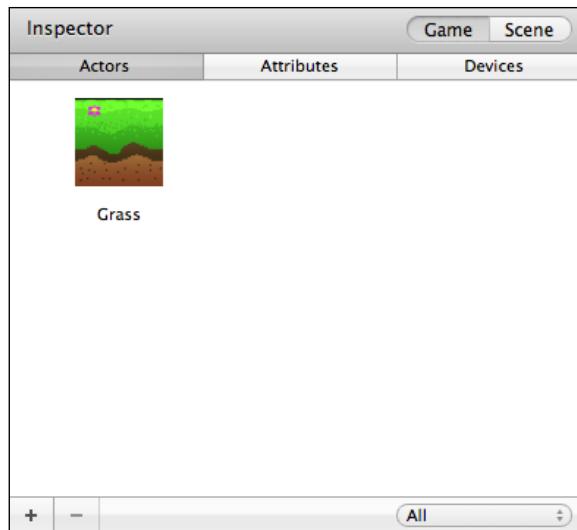


Under the **Library** section, click on the **Media** button. This section will show you all your imported images. Click on the **+** button at the bottom of the **Library** window, and a window will open to select your media. Simply select all the images you want to import and click on **Open**, and that's it! If you want, you can also drag-and-drop your image files directly into the **Library** window for fast importing. The same goes for importing sound effects and music, but you click on the **Sounds** button instead.

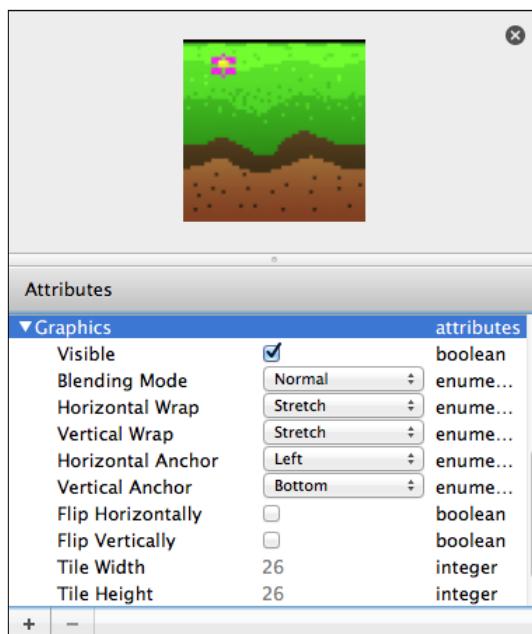


All imported! Now let's start laying out a quick level, we won't do any behaviors just yet.

To create an actor (the quick way), simply drag the image into the **Actors** section under the **Inspector**. GameSalad will automatically rename the actor with the image name. If you want, you can rename it by clicking on the actor's name.



Now, click and drag that actor into the level window and place it wherever you like. We want to stretch this actor so it tiles all the way across the level, as opposed to having to create hundreds of actors to make up one platform. So double-click on the Grass actor (I named it `Grass` but you can name it whatever you like), under the **Attributes** section, scroll down to find the **Graphics** drop-down list, and under **Horizontal Wrap**, click on **Tile**.



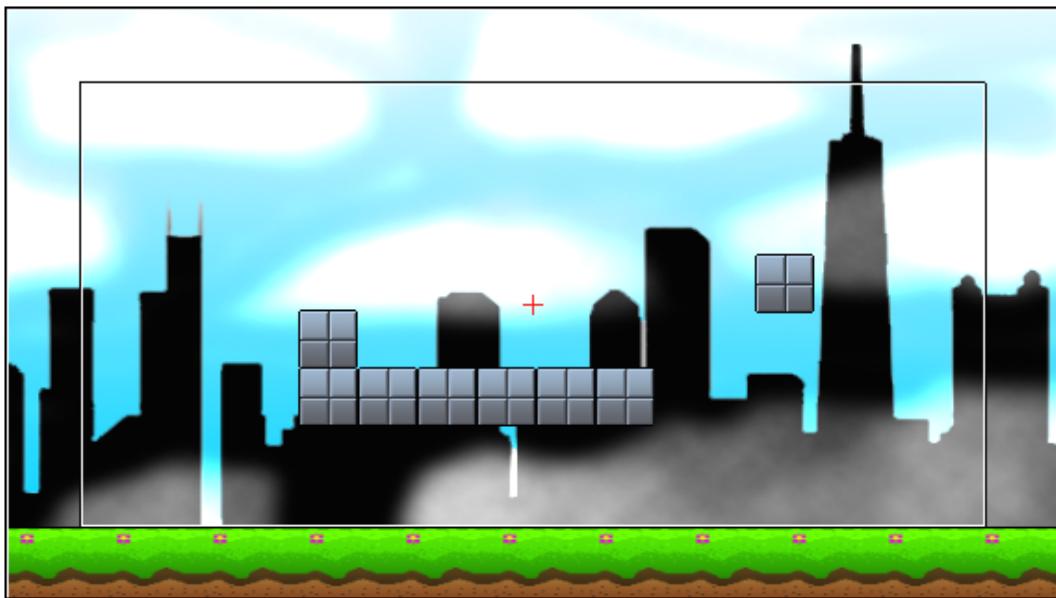
Now go back to our scene, click on the actor within our level and stretch it sideways. It will now repeat the same image over and over as you stretch it horizontally. You can do this both horizontally and vertically to create massive walls, or even buildings!



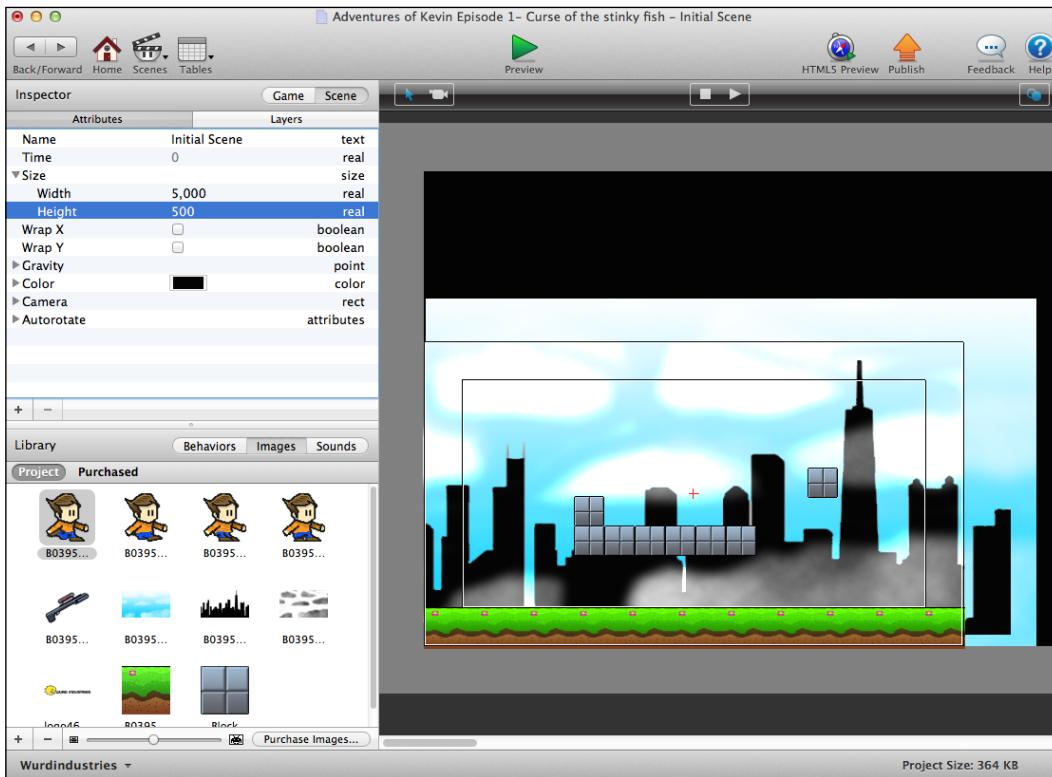
Don't forget that you can use the navigation buttons to easily navigate windows in GameSalad!



Go ahead and start populating your level. Uh oh! What's that? An actor is covering up another one? No problem! Right-click on the actor that's covering up the other, and click on **Send to Back**. Yes, you have layers within the scene editor; you can move objects back and forth to get the desired depth. If you want to select an actor that is behind an object, simply click on the **Scene** button next to the **Inspector**, and expand the current layer. You'll now have all the objects within the scene, you can select the one you want and even move it up and down through the layers by clicking and dragging for more precise placement.



Hmm... Not a lot of space though, is there? No problem! We can extend the scene size.



Click on the **Scene** button in the **Inspector** window and open the **Size** drop down. Bingo! Simply extend the level to the size you like. I like it a little bigger height-wise as well so it doesn't look so confined when scrolling. Much better! Now you can continue building your complete level.

[

**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.

]

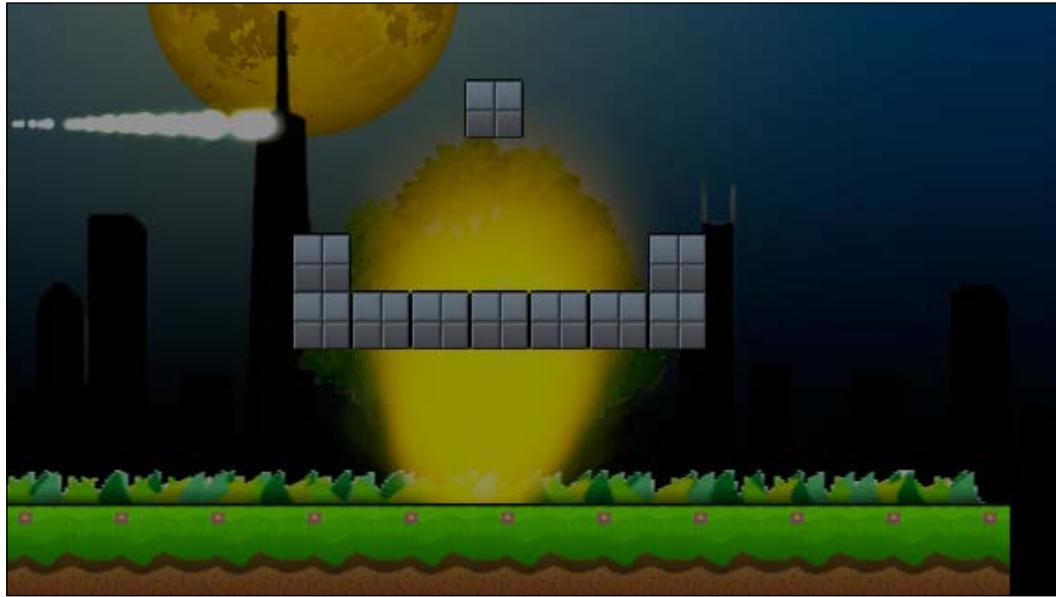
Adding a lot of little details certainly adds to the environment. For example, you can add some trees in the background, or some swaying grass in the foreground. All these little things really add to the gaming experience.



Here's an example of a level that you can create in GameSalad. I know you are going to make an awesome level! Pretty simple to create levels, isn't it? This really is how simple GameSalad is, pretty much all dragging and dropping. We aren't done yet though. It's only the second chapter; we still have a lot to cover.

Why not try laying out a few levels? Simply go back to the **Scenes** tab and click on the **+** button at the bottom of the screen to create a new scene.

For the game we are going to create, our hero Kevin returns to his home town only to find it is cast in darkness, is in ruins, and reeks of fish. The next figure is what he will return to. It's not finished yet, but it sets the mood.



Those levels you designed in the design document, or just sketched down on a pad of paper, can be realized this easily! The great thing about GameSalad is you get a full view of the level, and you can put your level together within minutes. It really is a fantastic piece of software.

I hope you enjoyed this chapter, and that you learned a lot about the game designing process. Sit tight, this journey is about to get pretty crazy!

## Summary

In this chapter, we discussed a lot about the game design process. Some good procedures to follow and how it can benefit you during the development process. These are just recommendations for you, as you can tweak your own design process to whatever works for you. We also discussed how you can use certain design documents, and how you can take your drawings and put them into GameSalad. From there, we learned how to start designing levels within GameSalad.

In the next chapter, we are going to delve a little deeper into GameSalad. We're going to discuss how to create a main menu, game introductions and logo animations, option menus, and level selection menus.

Go get yourself a nice coffee, beer, glass of wine, or milk, and let's dive into the next chapter.

# 3

## Diving In – Introductions and Menus

How many games have you switched on? The first thing you see is the company's (on most cases, companies) logo(s)? From there, it goes into an introduction animation, and then on to the main menu. In this chapter, we are going to discuss the following:

- Logo animation
- Creating main menu
- Introducing animation

Don't worry, we are going to get into the gameplay soon! First, people have to see who created this masterpiece! On to Logo animation!

### Logo animation

The one unfortunate thing about GameSalad is that it doesn't yet allow videos to be imported. This isn't a problem though; because we can do all sorts of animations within GameSalad and have them look pretty good. To start off, we are going to do a very quick logo animation. I know it's not the most exciting topic, but it does come in handy.

Here is the logo for my company, Wurd Industries:



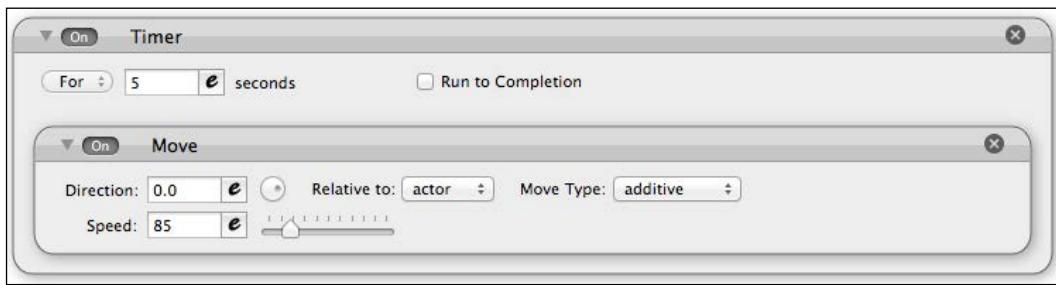
We could do a simple splash screen that shows this logo, but I don't really think that would fit into the theme of the game that we are creating. What we'd like to do instead is have our hero, Kevin, walk into a cavern or a dungeon, open up a treasure chest and find the logo inside it.

This kind of logo animation will require quite a few different elements and behaviors. Create a **New** scene in GameSalad, and call it `1ogo`, or something to that effect. For now, let me show you the scene he is going to walk into:



Nothing spectacular, but once it's animated, it will look awesome! Let's start animating this scene. Let's double-click on our Kevin actor, then click on the big lock to edit the behaviors of this actor. By doing this, we edit only this actor, not the main actor in the **Inspector** panel. This allows us to use the same actor over and over, and give each one separate behaviors without making changes to the main actor.

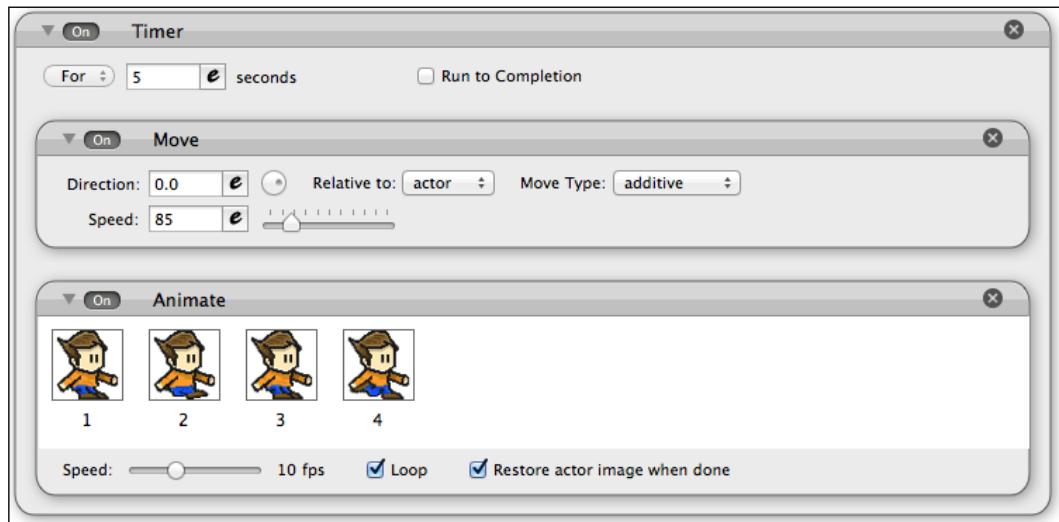
 Don't forget to download all the supplied resources for the book and drag them into your library so you can follow along. Now we are going to make him move. Under **Library** | **Behaviors**, scroll down to **Timer**, then click and drag the behavior into the behavior section.



Now click the **Every** drop down, and change it to **For**. We are going to leave the **seconds** at 5. Now scroll down and find the **Move** behavior. When you find it, drag and drop it into the **Timer** behavior. Change the **Speed** to 85. You may need to test this for yourself depending on where you want him to end up. You can slow him down and increase the timer, or vice versa. It all depends on how you want the scene to look.

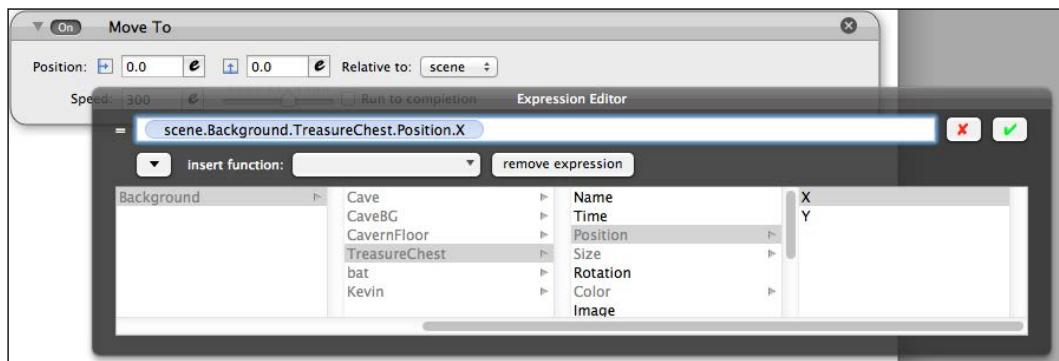
 You can drag in almost any behavior into a timer. There's a lot of flexibility here because you can have the behavior run for a certain amount of time, have it run every set amount of time, or have it run after a certain time.

Now, click the play button and see how it looks, and if he walks to where you want him to. He does? Great! Only one problem, he's walking but not animating. No problem, let's drag in an **Animate** behavior into the timer. From here, drag in the walking images to form the animation. Again, you will have to play around with the speed to make sure it looks right.



Check your behaviors with the preceding image. You should have a timer set for 5 seconds, a **Move** behavior, with the direction set to 0, and speed set to 85, then an **Animate** behavior with our walking animation.

If everything works, you can do the same for any background element you may have in the scene, such as the bat that I added in the scene. But instead of having a timer, just have a Move and Animate behavior so it's constantly moving, and animating. You can also use a **Move To** behavior to make the actor move to a specific location, or object.



Simply clicking on the **e** button will bring up the expression editor. From there, you can select a specific location you would like the actor to move to. We will look into this a little deeper later on. Now would be your chance to add music files to the actor. Simply import a music file the same way you would import an image (by clicking and dragging it into the Library, or by clicking on the **+** button), then you drag the sound file into the behavior section and it will automatically add a **Play Music**, or **Play Sound** behavior.

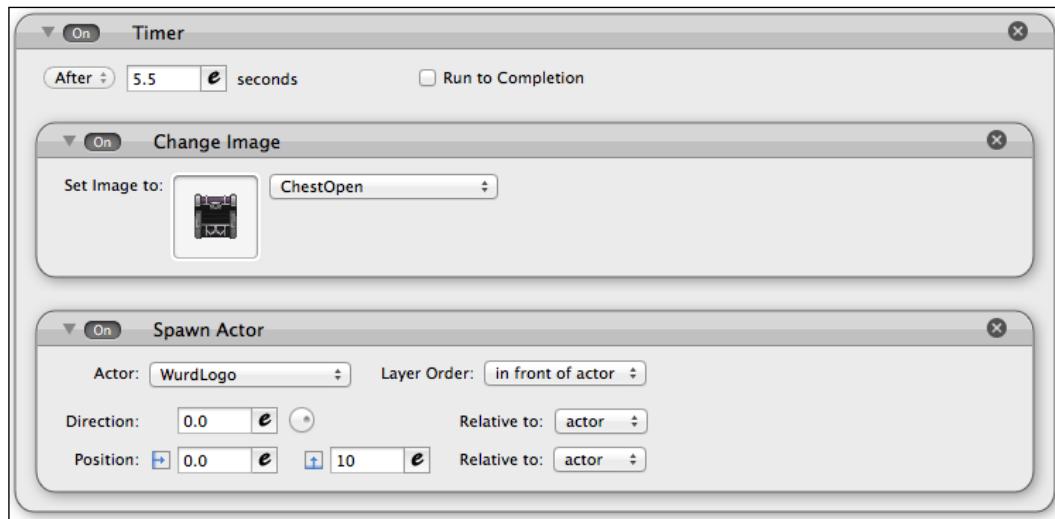


Now let's edit the treasure chest. Again, we are going to edit the actor in the scene, not the one in the **Inspector** panel. Drag in a Timer behavior, change it from **Every** to **After**, then change the value to 5 . 5 seconds. Now we aren't going to animate this chest, we are just going to change the image. You can simply find the image you want to use and drag it into the **Timer** behavior. This will automatically create a **Change Image** behavior.

For the logo, I split it up so I just have the logo (seen in the following figure), not the text, scaled it down to 32 x 32 and created a separate actor for it.



Now let's spawn the logo. In the behaviors, find **Spawn Actor** and drag it into the **Timer**.



I changed the vertical position to 10, which would be 10 pixels above the chest.

Let's go to the logo actor. We can use the one in the **Inspector** panel because this is probably the only time we are going to use this actor in the game.

Create a Timer behavior, and change its **After** text field to 1 second, then drag in a **Move To** behavior. Click on the **e** beside the horizontal position. Click on the down-arrow button to expand the editor. Select **Game | Display Size**, then double-click on the **Width**.

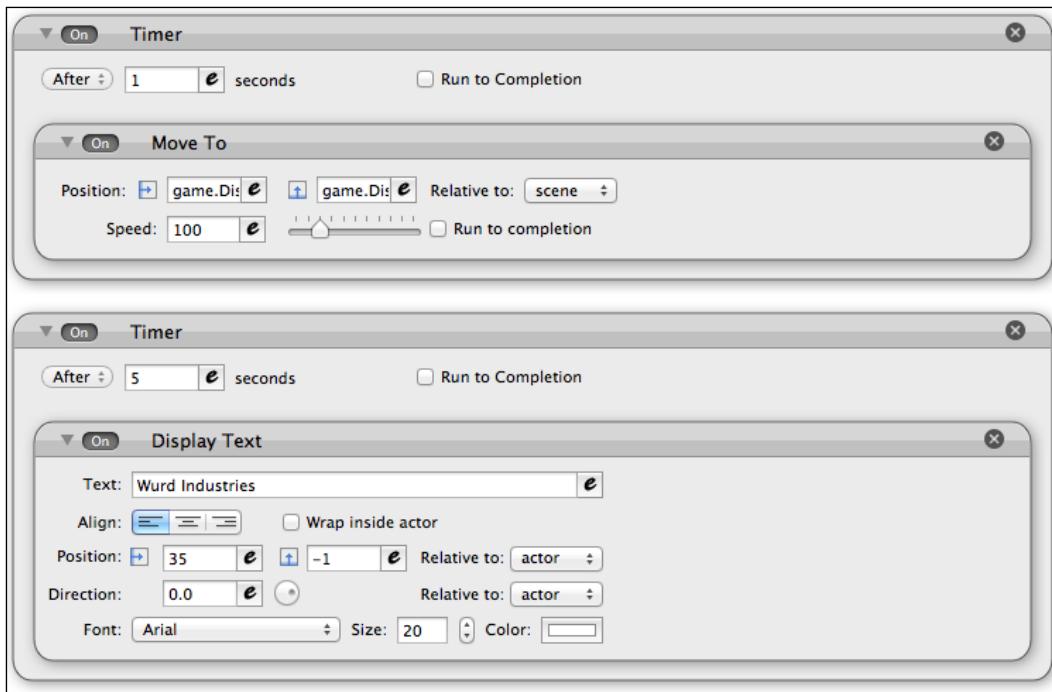
It will now show up above in text as **game.displaySize.Width**. After that expression, type in **/2 - 100**. This will place the actor in the middle of the screen, minus 100 pixels.

Do the same for the vertical position, but instead of **Width**, you'll choose **Height**; then type in **/2**, to divide by 2.

Both expressions should be written out like this:

- **Horizontal:** `game.Display Size.Width /2 -100`
- **Vertical:** `game.Display Size.Height /2`

Then set the **Speed** to 100. For reference, take a look at the following image to compare your behaviors:



Then drag in another separate Timer behavior, change its **After** field to 5 seconds (the time it will take for the logo to move to its spot), then drag in a **Display Text** behavior.

- **Text:** Wurd Industries (Or whatever the name of your company is, or your name)
- **Position (horizontal):** 35
- **Position (vertical):** -1
- **Font:** Arial
- **Size:** 20
- **Color:** White

Of course, these are the values I used. You can use whatever you like for your game. Now let's test out the level to make sure everything works in it.



Now we simply have to change the level after a few seconds, go back to the logo actor, add in a Timer behavior, change it to **After 6.5 seconds**, and then drag in a **Change Scene** behavior. This will automatically show the next scene, but you can change it to the previous level, or even one of your choices. As you can see, there are a few elements missing from the original scene I showed you. I moved them all to the furthest back layer so I had easy access to the actors in the scene.

That's all for our logo animation. Again, I know it's not that exciting, or even required, but it will come in handy at some point.

Now let's create the main menu!

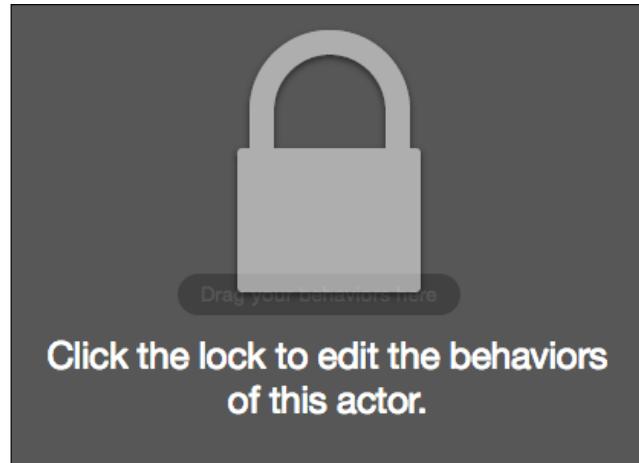
## Creating the main menu

Now let's go back to our project's home screen, and add a new scene. This will be our main menu, so name it accordingly. There are a few elements here that you can create separately, such as the menu text or game logo, menu buttons, and any kind of background element (if you haven't created them already).

For my menu, I'm reusing the cave background from the previous level, and I also created a logo and a button image. Create new actors with the images you imported (simply clicking and dragging the image into the library will create the actor from the image) and arrange them within the scene.



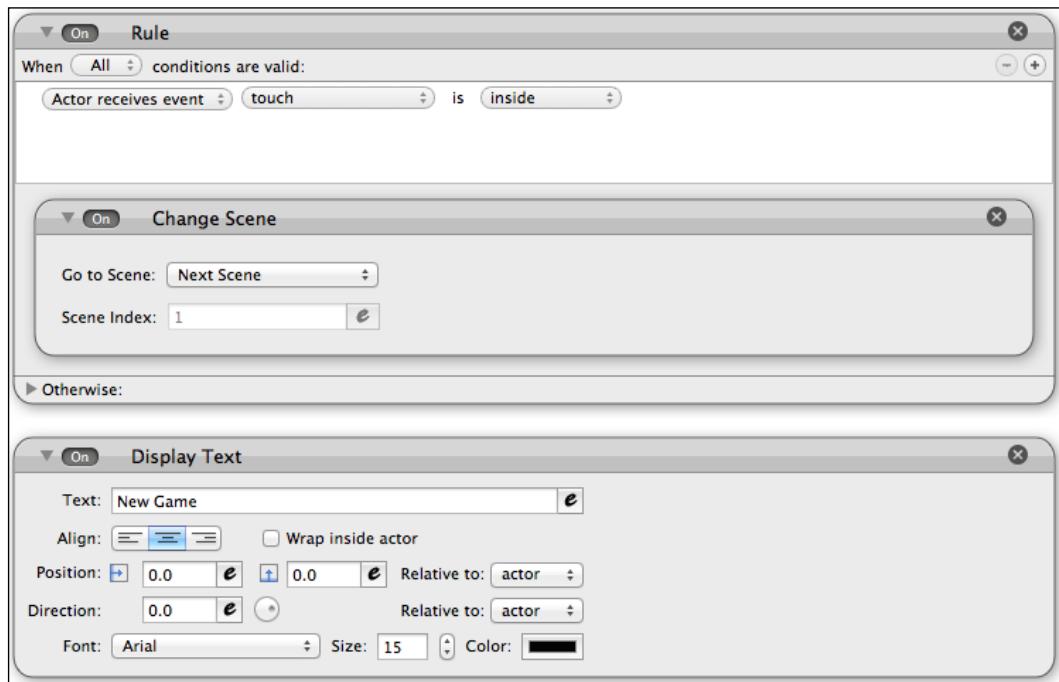
This is what I created. Now we are going to work on these buttons. Double-click on the first button, and then click on the large lock (seen in the image below) to edit this actor only. Now click on the **Create Rule** button at the top right of the screen.



You will now be presented with a **Rule** window. Rules are exactly what they sound like; from here, you can program key strokes, button presses, collisions, and more. The default settings are **Actor receives event | mouse position | is | inside**. We want to change this, so click on the **mouse position** drop down, and select **Touch** and change it to **is inside**.

Now find the **Change Scene** behavior and drag it into the rule. Now, when you click on the top button, you will move to the next scene. We need to add a label though, so let's add in a **Display Text** behavior.

This will be our new game button, so change the text to **New Game**. For the image I created, the text fits perfectly at a size of **15** with black color, but you can change those values to whatever you want.



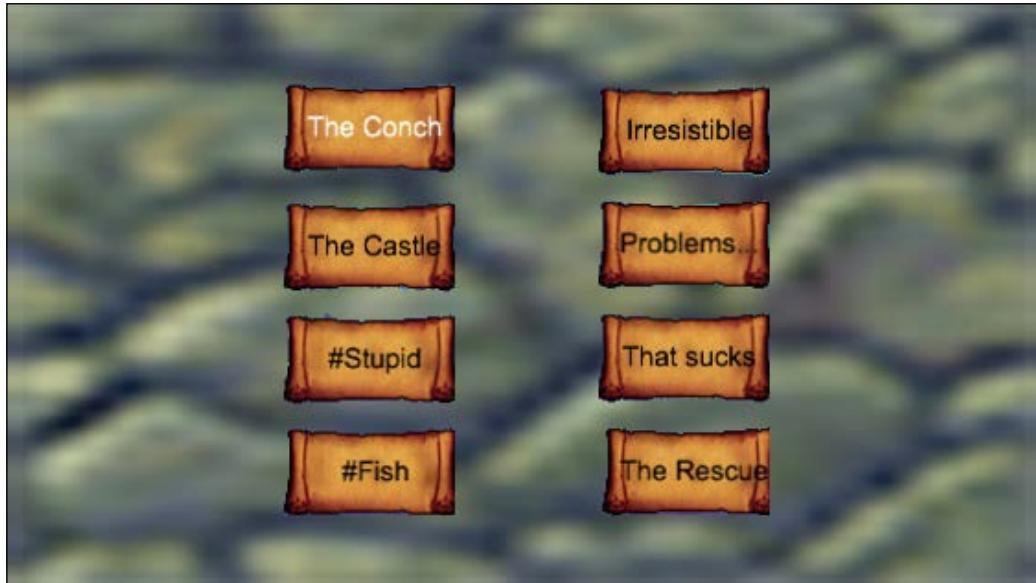
We will do the same for the second button, so hold *Shift* + click on the two behaviors here (the display text and rule) and copy them by pressing *Ctrl + C* (Windows), or *Command + C* (Mac). Now double-click on the second button, and then click on the lock and paste the behaviors by pressing *Ctrl + V* or *Command + V*. Change the text to **Continue**, and the **Change Scene** behavior from **Next Scene** to **At Index**. We are going to create an integer that will save the player's current level, and when we click on the **Continue** button, it will jump back to that level. Let's go back to our scene, and in the **Inspector** section, select the **Attributes** tab, and click the **+** button. A pop up will now display asking what type of attribute you want to create. Select **Integer** and click on **Choose**. Double-click on the name and change it to **Saved Level**, or whatever you will remember.

Now test your game to see if the buttons work; the default integer for our **Saved Level** is zero, so when you click on the **Continue** button nothing will happen. You can check to see if it's working by changing that number from 0 to 1. Now when you click on the button, it should go back to the first scene.

## Selecting levels

For this game, I'm not going to include a level selection menu. However, if you want to create one, let me show you.

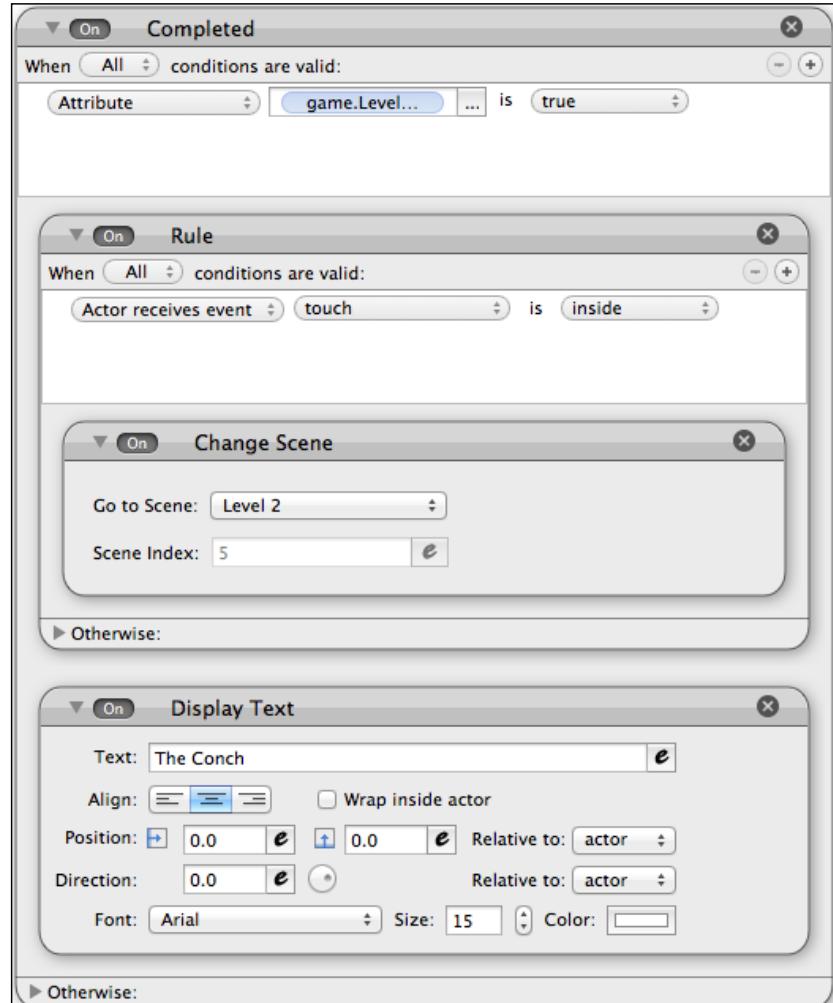
Let's start by creating a new scene. I'm going to start by arranging a bunch of buttons in the scene. We will follow the same formula we used on the main menu, but each button will load to a different level:



You can simply copy and paste the main menu button behaviors and simply change the level to the one you want to load to, or you can do an unlocking system like I did. Here's how to do it:

1. In your scene, create a new attribute in the **Inspector** section under the **Attributes** tab, make it a **Boolean**, and name it according to the level it will unlock, for example, `Level1Complete`.
2. Now open up the first button (or whichever button you want), and let's create a new rule. Change it from **Actor receives event** to **Attribute**. In the box beside the drop down, click on the ellipsis (...) button, this will open up the **Attribute Browser** section.
3. Then click on **Game | Level1Complete** or whatever you named it, and make sure it's set to **is | true**. To make things a little more organized, let's name this rule so we can distinguish it from the other one we will create. Double-click on the **Rule** title, and rename it to `Completed`.
4. Within that rule, you can paste in the main menu behaviors, or just recreate them by creating a new rule inside of that rule (by clicking the **Create Rule** button at the top of the screen, and clicking and dragging it inside the `Completed` rule).
5. Change this rule to **Actor receives event | touch | is | inside**, then drag in a **Change Scene** behavior inside.
6. Now drag in a **Display Text** behavior inside the `Completed` Rule, but not inside the **touch** rule we just created. Change the text to the name of the level, and change the color to white. (We are going to make disabled buttons have black text so you can tell it's disabled.)
7. Create a new rule outside of the `Completed` rule, and name this one `Disabled`, or `Unavailable`, or something to that affect. Change it to **Attribute | game.Level1Complete | is | false**. Then just drag in a **Display Text** behavior (or copy the last one), and change the text color.

When the player completes a level, just change that attribute to **true**. It's that simple!

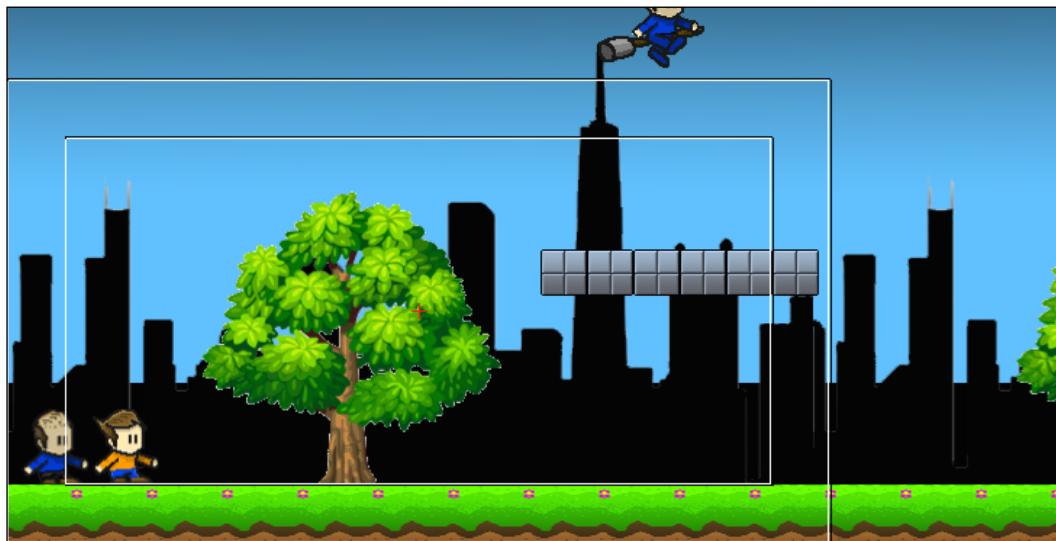


## Creating our introduction

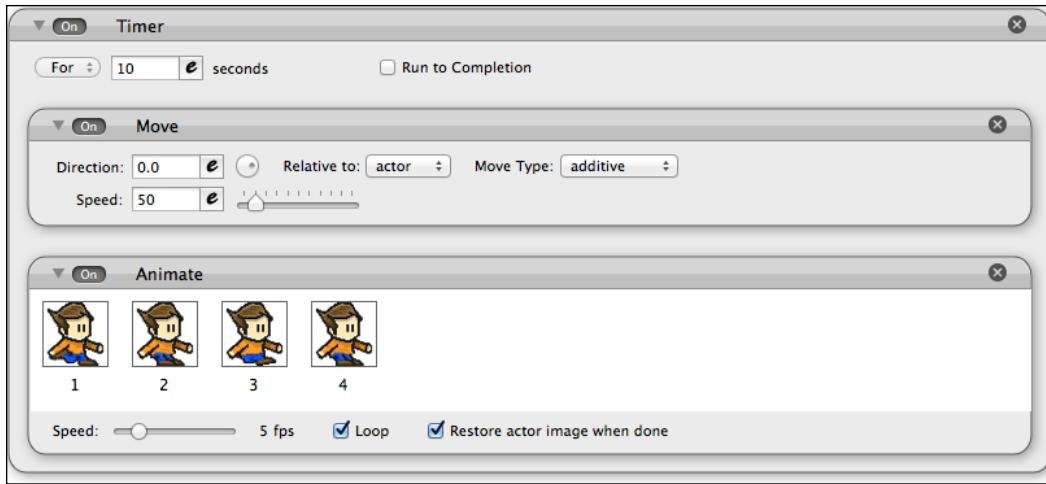
A game introduction sets the tone for the whole game, and they fill you in as far as the story is concerned. The introduction will use the same techniques we used in the logo animation, but we are going to go a little more in depth.

Let's start by creating a new scene right after our main menu.

The scene I'm going to create will have our hero Kevin, and his friend **Dominic** – who he has to save at the end of the game – walking across the screen, conversing. They will stop, then the evil wizard **Magellan** will appear from the top of the screen, kidnap Dominic, and lay a curse on the land. Kevin will then run after Magellan to save Dominic.

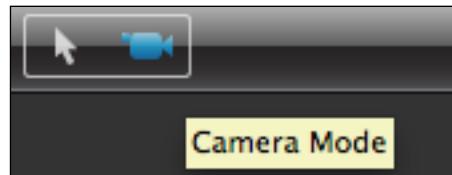


Here's the scene I did, just a simple little scene. I can add some more background elements later on. Let's open up our Kevin actor by clicking on it; again click on the big lock to edit just this actor. Now let's add in a Timer behavior, change it to **For | 10 seconds**. Now drag in a Move behavior and change the speed to 50. Again, you can change it so he moves faster for less time, or slower for more time, whatever looks right to you. I chose a little slower because there's going to be a short dialogue. Inside the Timer behavior, add in an Animate behavior and import the walking images from your Library. I changed the animation speed to 5 to match the slow walking speed. Now click on the Timer behavior and copy it, then go to the Dominic actor and paste it in. The only changes you need to make are in the walking images.



Now test the scene. They should walk in unison together. Do they just walk off the screen? We need to set up scrolling here. Double-click on the actor you want to center the screen on and find the **Control Camera** behavior. That's it!

You can also control the camera scrolling boundaries by going back to the scene and clicking on the **Camera** button at the top-left corner of the scene screen. (Ha! That rhymes!)



Now the scene screen will change; you will now see yellow boundaries light up. These are the boundaries the actor that controls the camera will have to leave before the camera scrolls. I always drag the handles to the center, so as soon as the actor hits the center of the screen, it starts scrolling. Using this camera mode, you can not only change the scrolling boundaries, but you can also move the camera around. To exit this mode, just click on the actor mode, which is the arrow icon beside the **Camera Mode**.

Now that the two characters are walking together, it's time for some dialogue. There are a few ways we could do this, such as a legend in **The Legend of Zelda**-style text boxes, or speech bubbles. For ease of use, I'm going to show you how to do a **The Legend of Zelda** style text box.

Here is the speech box I created. It's nothing complex, just black and white with a simple texture in the background:



[  Don't forget that you can access all the images and assets used in the game project in the resources section of this book. ]

Let's import this image into GameSalad, and make an actor with it. Place it in the scene in the middle close to the top. Don't worry about exact placement; we will force it to appear at dead center. Double-click on the text box in the scene and click on the lock to edit. Change the color to `0.75 Alpha` just so the player can see behind the text box a little bit. This may get a little confusing, but stick with it. I'll try to explain everything as much as I can. Click on the **Home** button, and click on the **Tables** tab. We are going to click on the `+` button to create a new table; name this table `Intro_Speech`. Double-click on it to edit it where it says `rows`, change it to the number of lines of speeches you would like. For me, I put `6`. Click the row `0`, column `1` drop-down box, and change it to `Text`. Now you can fill in the rows accordingly.

When done, your table should look something like this:

Intro_Speech		Rows 4	Columns 1
		1	
			Text
1		Dom: What a nice day Kevin!	
2		Kevin: It sure is Dom!	
3		Dom: *Sniff* Ugh! What's that smell?	
4		Kevin: It wasn't me!	

Now let's go back to the scene, and we are going to add in two new attributes: one `CurrentSpeech` (will be a text attribute), and the second one is `CurrentRow` (will be an Integer). Now let's shoot back to our text box actor and open it. This is where it may get a little confusing, but it saves so much time!

Drag in a **Change Attribute** behavior, change it to **Change Attribute | game.currentSpeech to | tableCellValue (game.Intro Speech, game.currentRow, 1)**.

Let's break this down. What you did here is create a function that immediately changes the **CurrentSpeech** attribute to the **Currentrow** attribute in the **Speech** table.

 You cannot type in these values! You must click on the ... button to change the attributes, then click on **Game**, then **CurrentSpeech**. Then click on the **To** and then the **e** button to open the expressions editor. To find the **tableCellValue()**, click on the **insert** function drop down box and scroll down near the bottom. When you click on it, it will show up like this: **tableCellValue(table,row,col)**. Highlight **table** within the brackets, and click on the down arrow button to fill in the attribute **game.intro Speech**, then highlight the row and do the same, but click **game.currentRow**. Finally, highlight **col** and type in the number 1. If you simply type these values in, it will not work.

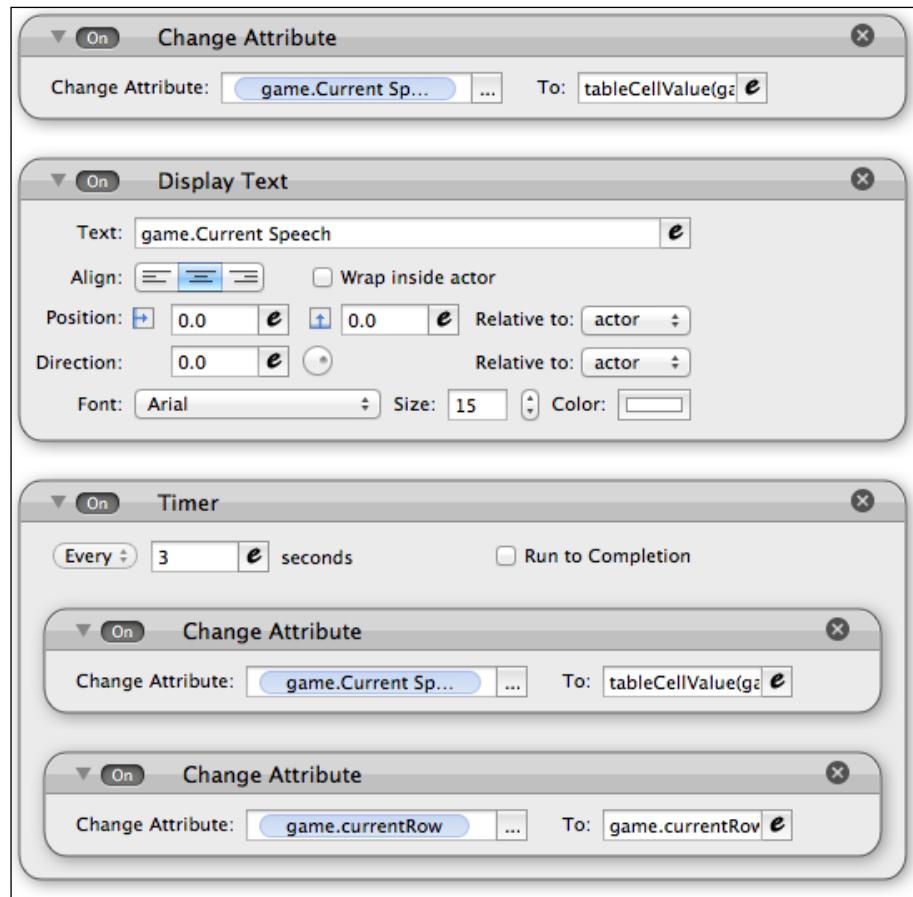
Now let's drag in a **Display Text** behavior, and click on the **e** button to change it to **game.Current Speech**. I changed the text size to 15, but use whatever works for you.

This will change the currently displayed text to the current row of text within the table (as defined in the previous image).

Drag in a **Timer** behavior and change it to **Every 3 seconds**, or for however long you want the text to delay changing to the next line. We are going to drag in two **Change Attribute** behaviors, as follows:

- **Change Attribute from game.Current Speech | to | tableCellValue (game.Intro Speech, game.currentRow, 1)**
- **Change Attribute from game.currentRow | to | game.currentRow + 1**

The first attribute will change the CurrentSpeech to the corresponding row within the table. The second attribute will move to the next row in the table.



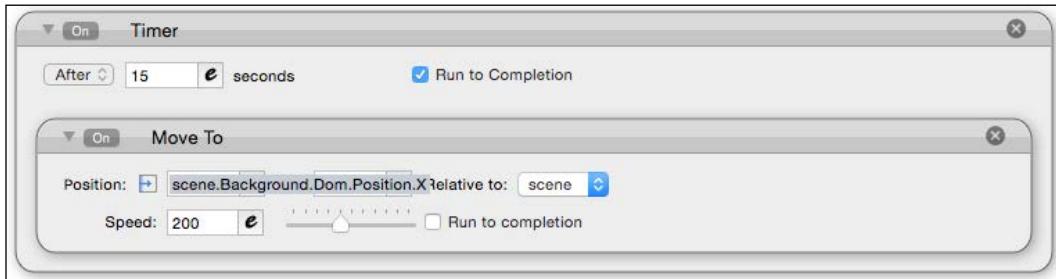
Now test it out to make sure that everything works perfectly and the text changes properly. If not, go back to make sure everything was input properly.

If you are having a hard time with tables, check out the official GameSalad cookbook's tutorial for tables at: <https://help.gamesalad.com/hc/en-us/articles/202044496-2-6-Tables>.

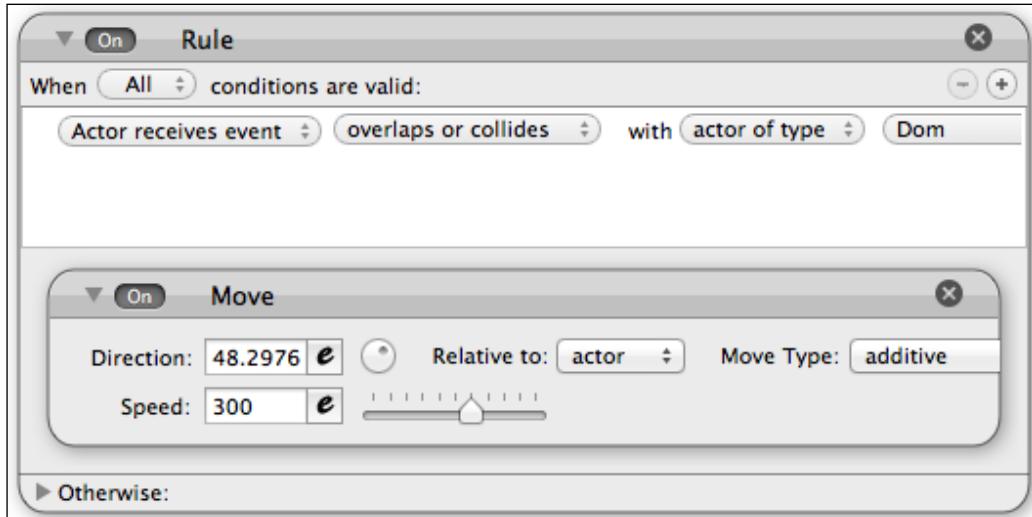
If the conversation is working, let's go to our wizard moving. If you haven't created him yet, do that now and then drag him into your scene, out of the camera's view.

Double-click on him and click on the lock icon. I timed out when the actors stop walking, and when the conversation stops, it's about 15 seconds. So, drag in a Timer behavior, and change it to **After 15 seconds**.

Inside the Timer behavior, drag in a Move To behavior, for the horizontal position. Change it to `scene.Background.Dom.Position.X`, and the vertical to `scene.Background.Dom.Position.Y`, and the speed to 200. Now we are going to create a collision rule so that when the wizard collides with Dominic, he will take off the screen super fast.



Click on the **Create Rule** button at the top right of the screen, and change it to **Actor receives event | overlaps or collides with actor of type | Dom**. When you click on the last drop-down box, it will show you a list of all the actors you have created. You can also create tags such as enemies, so you don't have to create behaviors for every enemy in the game. Instead, you can just tag each actor as an enemy and just have them colliding with the actor with **tag | enemy**. Convenient!



Drag in a Move behavior, set the direction to the upper-right direction of the screen, and set the speed to something crazy fast, like 300.

Test the level to see what happens. Ideally, Magellan should come down from the top, and as soon as he hits the Dom actor he should move away. But he just stops! He doesn't keep moving. Well, we can fix that. Double-click on the Dom actor, and we are going to create another collision rule, but the actor will be Magellan. Now drag in a Move To behavior and for the positions  $(x, y)$ , change it to `scene.Background.Magellan.Position.(x, y)` respectively, and set the speed to **300**, or the same speed with which you made Magellan speed off the screen. Now, as soon as Magellan collides with Dom, he will take off and the Dom actor will follow.



## Working with particles

I know what you're saying now, "Dude! Nice particles! How do I get such awesomeness?" Well, let me show you, my friend. Double-click on the Magellan actor in the scene, and drag in a **Particles** behavior. GameSalad's particle systems are pretty powerful, so there are a lot of variables we will have to change. Let's start on the first tab, **Spawn Rate**. For this effect, I changed the **Number of Particles** option to **100**, and changed the **Particle Lifetime** option to **1 second**.

Click on the **Velocity/Position** tab, and change the **Emitter Offset** variables to **-40** horizontal, and **-30** vertical, and set the **Speed** to **0**. We don't want the particles moving, we just want them to follow Magellan.

Click on the **Size** tab, and change it to **15**. On to the **Color** tab, I changed it to a teal, kind-of-blue flame color, and changed the **Blending** to **Screen**. Skip the **Rotation** tab, we don't need any rotation. On to the **Image** tab, let's select the **Particle Image**, which will give the particle a soft round shape. You can use this image for flames and smoke, as well just by simply changing the color.

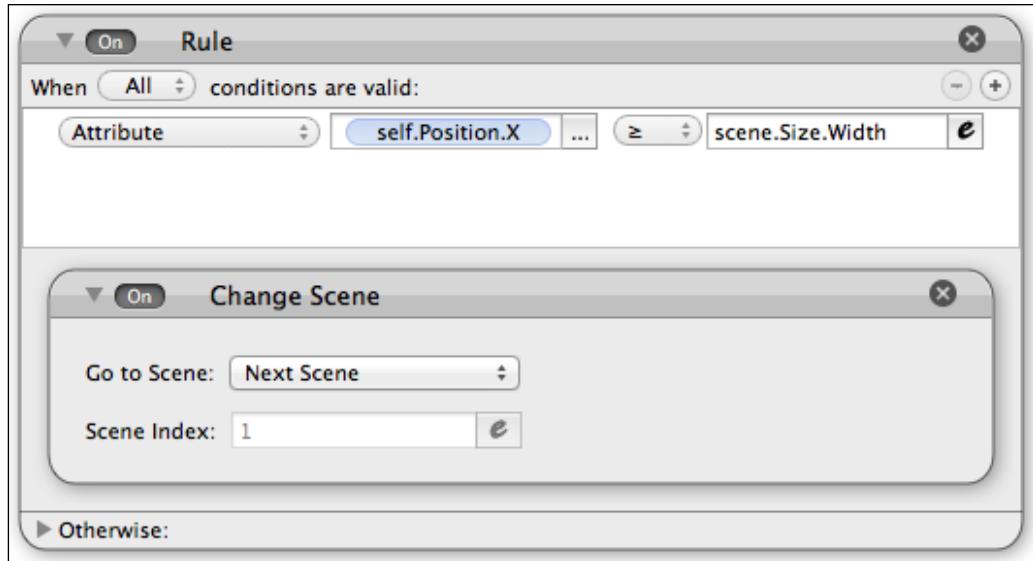
## Adding emotion to the scene

There are a few things in the scene that I added that we didn't discuss. For example, the background image has a **Play Music** behavior which has a nice, soft cheery tune. Then a timer after 8 seconds; it changes the color of the actor from the blue sky to an ominous purple. To do this, simply add in three Change Attribute behaviors and change each `self.color.blue(.green(.red))` to whatever value you like. Another thing I love about GameSalad: Even the color changing is powerful! After that, I added a new **Play Music** behavior to make it more ominous.

We now have to get our hero Kevin run off screen after Dominic gets kidnapped.

Open up our Kevin actor in the scene, and simply add in two Timer behaviors: the first one will be **After 18 seconds**, with a Move behavior running in the direction Magellan flew off, at a **Speed** of 200; and an Animate behavior with a faster frame per second value to make it look like he's running. The second timer will be **After 20 seconds**, with a Change Scene behavior to the next level.

You can also have it change levels when he reaches the boundaries of the scene. Simply add a **Rule** with **Attribute | self.Position.X | scene.Size.Width**, and drag in the Change Scene behavior:



All all, it's pretty simple to add some cool looking introductions in GameSalad. You can even expand on it! For example, after Kevin runs off the screen you can have the next scene with him running and seeing a big fish dropped on his house!



Yeah! Take this scene with a grain of salt. My craziness coming out there, but hey, it adds humor, and humor is fun! Think of Portal.

If you want to do some cool camera panning, you can do the same as we did with Kevin, but have a hidden actor (either `alpha` or uncheck `visible`), and have it move and control the camera. You can take these techniques we just used and twist them into your own interesting introduction animation. There are a lot of cool things you can do, and we are just scratching the surface! Sit tight, because now we are going to get into some gameplay.

## Summary

In this chapter, we discussed how to create a logo animation by spawning and moving actors, a main menu with working buttons, and a quick introduction to animation with moving actors, particles, conversations, and we had our first look at GameSalad's powerful tables. Don't worry though, we are going to look into them a little deeper further into this book. We also set up a few attributes that we will use further along in the book, such as the one for saving the levels.

Are you getting excited to continue? I know I am! Let's dive in deeper in the next chapter.

# 4

## Diving In – Player Creation

What game would be complete without a playable character? I can't really think of any game out there where you don't control some sort of character, or vehicle. It wouldn't be a game without some sort of playable object! It would just be, well, nothing really. We are going to create our playable character "Kevin", in this chapter. We are going to cover the following topics:

- Controls and player movement
- Collision detection
- Attacking

Now we get into the nitty-gritty fun of game development! By the end of this chapter, we are going to have a playable character that will do all we command.

### Controls and player movement

We are going to cover a couple of different control schemes, depending on the platform you want to release your awesome game on. We will deal with some keyboard controls, mouse controls, and even some touch controls for mobile devices.

Let's go into our project, and open up level 1.

1. First we are going to add some gravity to the scene.
2. In the scene editor, click on the **Scene** button in the **Inspector** window.
3. In the **Attributes** window, expand the **Gravity** drop down, and change the **Y** value to 900.
4. Now, on to our actor! We have already created our player actor in the Library, so let's drag him into our level.

5. Once he's placed exactly where you want him to be, double-click on him to open up the actor editor. Instead of clicking the lock button, click the **Edit Prototype...** button on the upper left-hand corner of the screen, just above the actor's image.

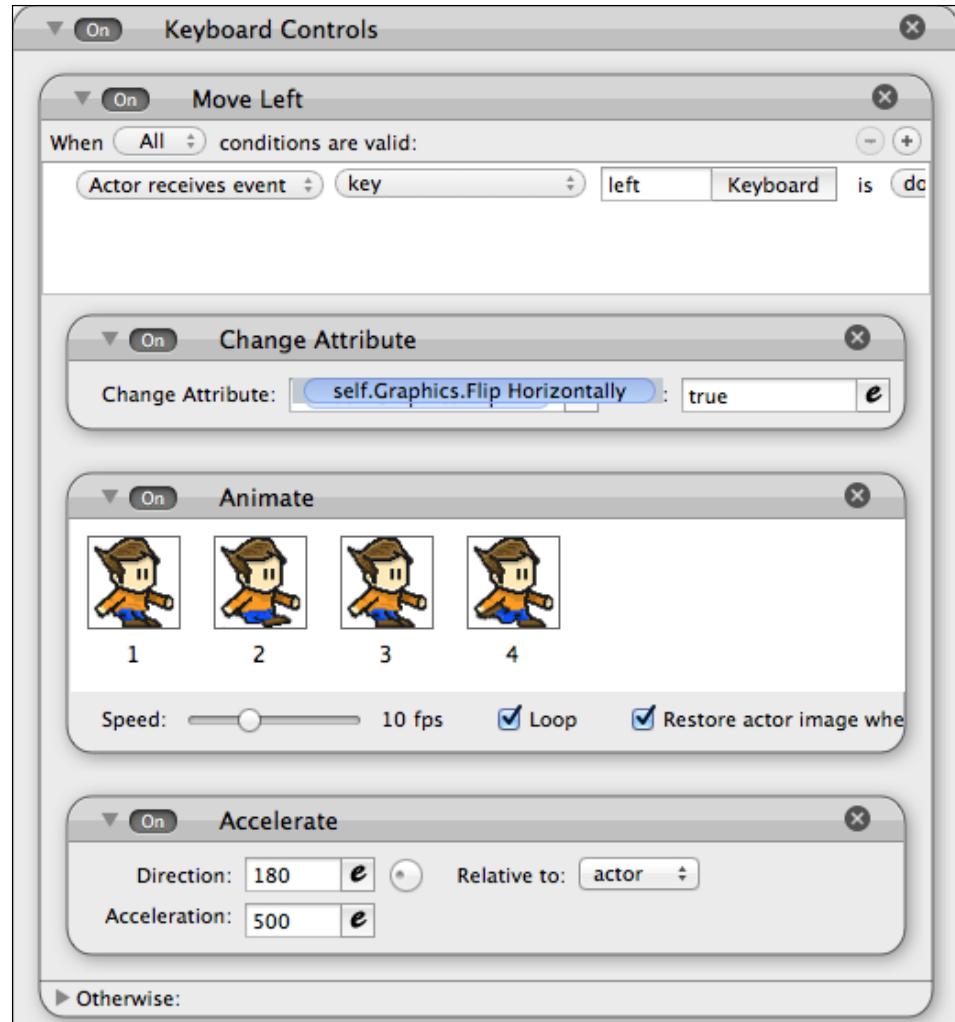


After doing this, we will edit the main actor within the Library; so instead of having to program each actor in every scene, we just drag the main actor in the library and boom! It's programmed.

For our player actor, we are going to keep things super organized, because honestly, he's going to have a lot of behaviors, and even more so if you decide to continue after we are done. Click the **Create Group** button, right beside the **Create Rule** button. This will now create a group which we will put all our keyboard control-based behaviors within, so let's name it **Keyboard Controls** by double-clicking the name of the group.

Now let's start creating the rules for each button that will control our character. Each rule will be **Actor receives event | key | (whichever key you want) | is down**. To select the key, simply press the Keyboard button inside the rule, and it will open up a virtual keyboard. All you have to do is click the key you want to use.

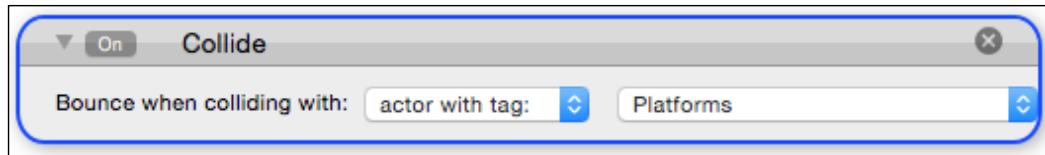
For our first rule, let's do **Actor receives event | key | left | is down**. Because our sprite has been created with Kevin facing the right side, we have to flip him whenever the player presses the left arrow key. So, we have to drag in a Change Attribute behavior, and change it to **Change Attribute: self.Graphics.Flip Horizontally to true**. Then drag in an Animate behavior for the walking animation, and drag in the corresponding walking images for our character. Now let's get Kevin moving! For each Move rule you create, drag in an **Accelerate** behavior, change it to the corresponding direction you want him to move in, and change the **Acceleration** value to 500. Again, you can play around with these moving values to whatever suits your style.



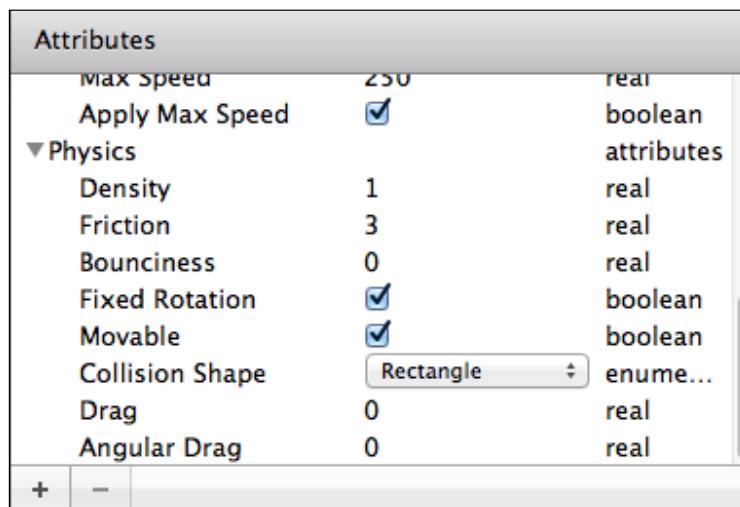
When you create the `Move Right` rule, don't forget to change the **Flip Horizontally** attribute to **false** so that he flips back to normal when the right key is pressed.

Test out the level to see if he's moving. Oops! Did he fall through the platforms? We need to fix that! We are going to do this the super easy way. Click on the home button and then to the **Actors** tab. On the bottom-left corner of the screen, you will find a **+** button, click on it. This will add a new group of actors, or tags; name it **Platforms**. Now all you have to do is, simply drag in each platform actor you want the character to collide against in this section. Now when we create our Collision behavior, we only have to do one for the whole set of platforms, instead of creating one per actor. Time saved, money made! Plus, it keeps things more organized.

Let's go back to our Kevin actor, and outside of the controls group, drag in a **Collide** behavior, and all you have to do is change it to **Bounce when colliding with | actor with tag: | Platforms**. Now, whenever he collides with an actor that you dragged into the **Platforms** section, he will stop. Test it out to see how it works.



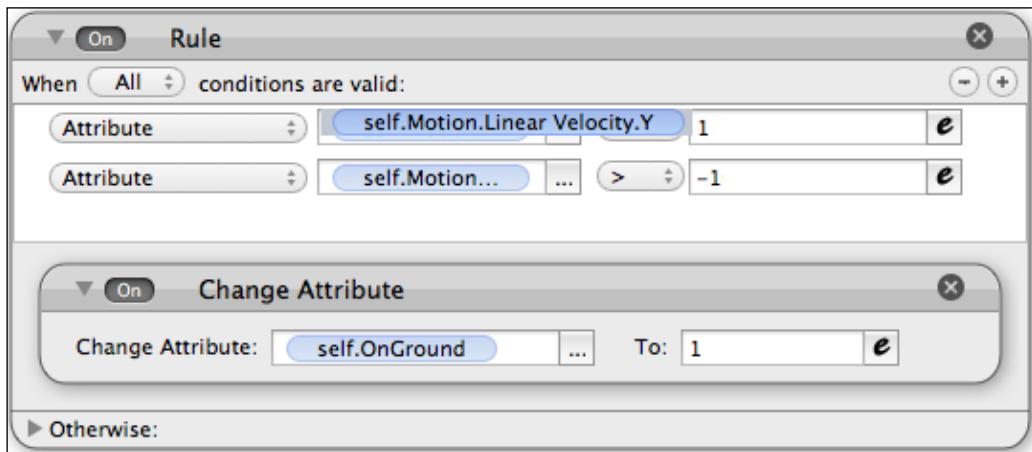
Bounce! Bounce! Bounce! We need to adjust a few things in the **Physics** section of the Kevin actor, and also the Platform actors. Open up one of the Platform actors (it doesn't matter which one, because we are going to change them all), in the **Attributes** window, open up the **Physics** drop down, and change the **Bounciness** value to 0. Or if you want it to be a rubbery surface, you can leave it at 1, or even 100, whatever you like. If you want a slippery platform, change the **Friction** to a lower than one value, such as 0 . 1. Do the same with our Kevin actor. Change his **Bounciness** to 0, his **Density** to 100, and **Friction** to 50, and check off (if not already) the **Fixed Rotation**, and **Movable** options. Now test it out. Everything should work perfectly!



If you change the Platform's **Friction** value and set it much higher, our actor won't move very well, which would be good for super sticky platforms. We are now going to work on jumping. You want him to jump only when he is colliding with the ground, otherwise the player could keep pressing the jump button and he would just continue to fly; which would be cool, but it wouldn't make the levels very challenging now, would it? Let's go back to editing our Kevin actor. We need to create a new attribute within the actor itself, so in the **Attributes** window, click on the **+** button, select **Boolean**, and name it **OnGround**, and leave it unchecked. Create a new rule, and change it to **Actor receives event | overlaps of collides | with actor with tag: | Platforms**. Then drag in another rule, change it to **Attribute | self.Motion Linear Velocity.Y | < 1**, then click on the **+** button within the rule to create another condition, and change it to **Attribute | self.Motion Linear Velocity.Y > -1**.

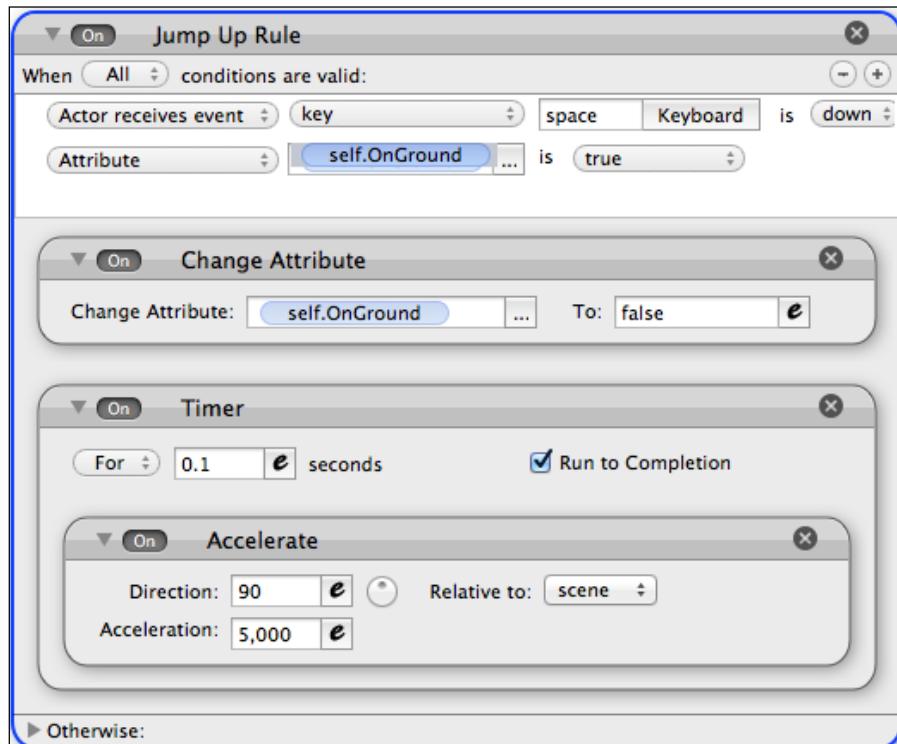
Whoa there cowboy! What does this mean? Let's break it down. This is detecting the actor's up and down motion, so when he is going slower than 1, and faster than -1, we will change the attribute.

Why do this? It's a bit of a fail safe, so when the actor jumps and the player keeps holding the jump key, he doesn't keep jumping. If we didn't add these conditions, the player could jump at times when he shouldn't. Within that rule, drag in a Change Attribute behavior, and change it to **Change Attribute: | self.OnGround To 1**. (1 being true, 0 being false, or you can type in true or false.)



Now, inside the Keyboard Controls group, create a new rule and name it Jumping. Change the rule to **Actor receives event | key | space | is down**, and we need to check another condition, so click on the + button inside of the rule, and change the new condition to **Attribute | self.OnGround is true**. This will not only check if the space bar is down, but will also make sure that he is colliding with the ground. Drag in a Change Attribute behavior inside of this Jumping rule, and change it to **self.OnGround to 0**, which, you guessed it, turns off the OnGround condition so the player can't keep jumping! Now drag in a Timer behavior, change it to **For | 0.1** seconds, then drag in an Accelerate behavior inside of the Timer, and change the **Direction** to 90, **Acceleration** to 5000.

Again, play around with these values. If you want him to jump a little higher, or lower, just change the timing slower or faster. If you want to slow him down a bit, such as walking speed and falling speed (which is highly recommended), in the **Attributes** section in the **Actor** window, under **Motion** change the **Max Speed** (I ended up changing it to 500), then click on **Apply Max Speed**. If you put too low a value, he will pause before he jumps because the combined speed of him walking and jumping is over the max speed so GameSalad will throttle him down, and that doesn't look right.



For the finishing touch, let's drag in a Control Camera behavior so the camera moves with Kevin. (Don't forget to change the camera's bounds in the scene!)

Now Kevin should be shooting around the level like a maniac!

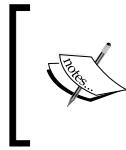


If you want to use mobile controls, let's go through some details. If not, skip ahead to the next heading.

## Mobile controls

Mobile controls in GameSalad are relatively simple to do. It involves creating a non-moveable layer within the scene, and creating actors to act as buttons on screen. Let's see what we're talking about.

Go into your scene. Under the **Inspector** window, click on the **Scene** button, then the **Layers** tab. Click on the **+** button to add a new layer, rename it to **Buttons**, and uncheck **scrollable**.

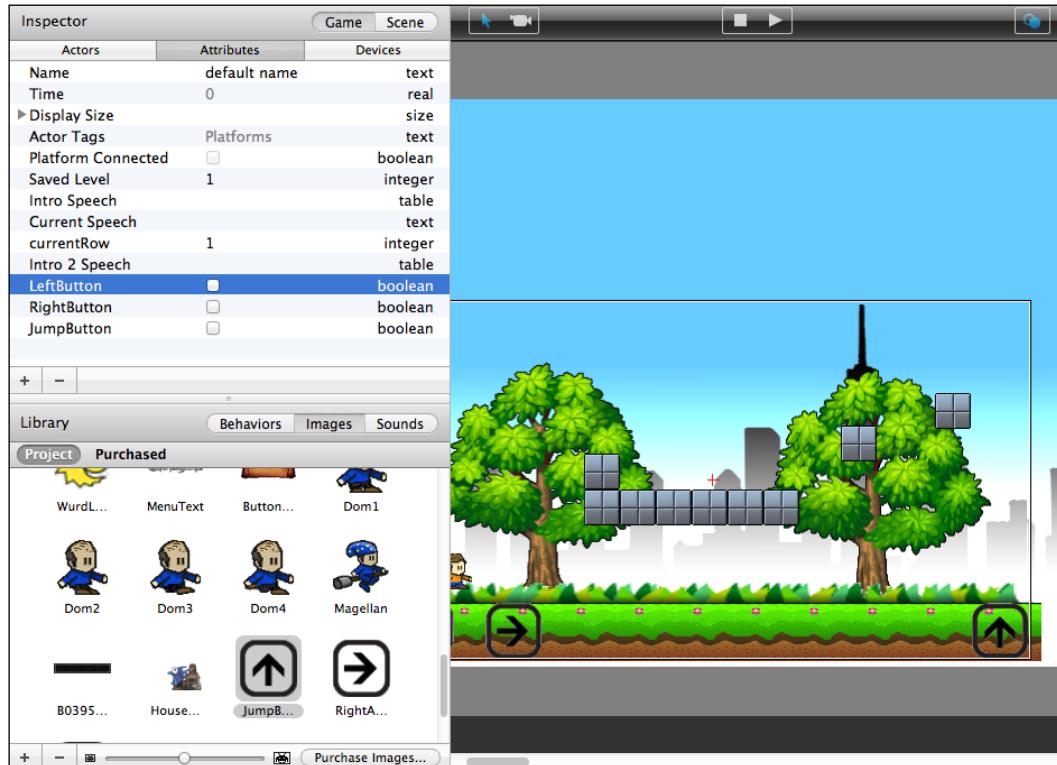


If you have a hard time renaming the new layer by double clicking on it, simply click on the layer and press Enter or the return key, you will now be able to rename it. Whether or not this is a bug, we will find out in further releases.

## Diving In – Player Creation

---

I've included three buttons for use within the resources section of this book for you to use in your project. Import these into your project, and create separate actors for each button. In each button, you'll want to uncheck the **Moveable** option in the **Physics** section, so the button doesn't fall off the screen when you click on play. Drag them into your scene in the ways you want the buttons to be laid out.



We are now going to create three new game attributes; each will correspond to the button being pressed. Go to our scene, and click on the **Game** button in the **Inspector** window, and under the **Attributes** tab add three new Boolean attributes, `LeftButton`, `RightButton`, and `JumpButton`.

Inspector		Game	Scene
Actors	Attributes	Devices	
Name	default name		text
Time	0		real
► Display Size			size
Actor Tags	Platforms	Ene...	text
Platform Connected	<input type="checkbox"/>		boolean
Intro Speech			table
Current Speech			text
currentRow	1		integer
Intro 2 Speech			table
LeftButton	<input type="checkbox"/>		boolean
RightButton	<input type="checkbox"/>		boolean
JumpButton	<input type="checkbox"/>		boolean

Now let's start editing each button in the Library, not in the scene. We'll start with the left arrow button. Create a new rule, name it `Button_Pressed`, change it to **Actor receives event | touch | is pressed**. Inside that rule, drag in a change attribute behavior and change it to `game.LeftButton | to true`; then expand the **Otherwise** drop down in the rule and copy and paste the previous change attribute rule into the **Otherwise** section and change it from **true** to **false**.

## Diving In - Player Creation

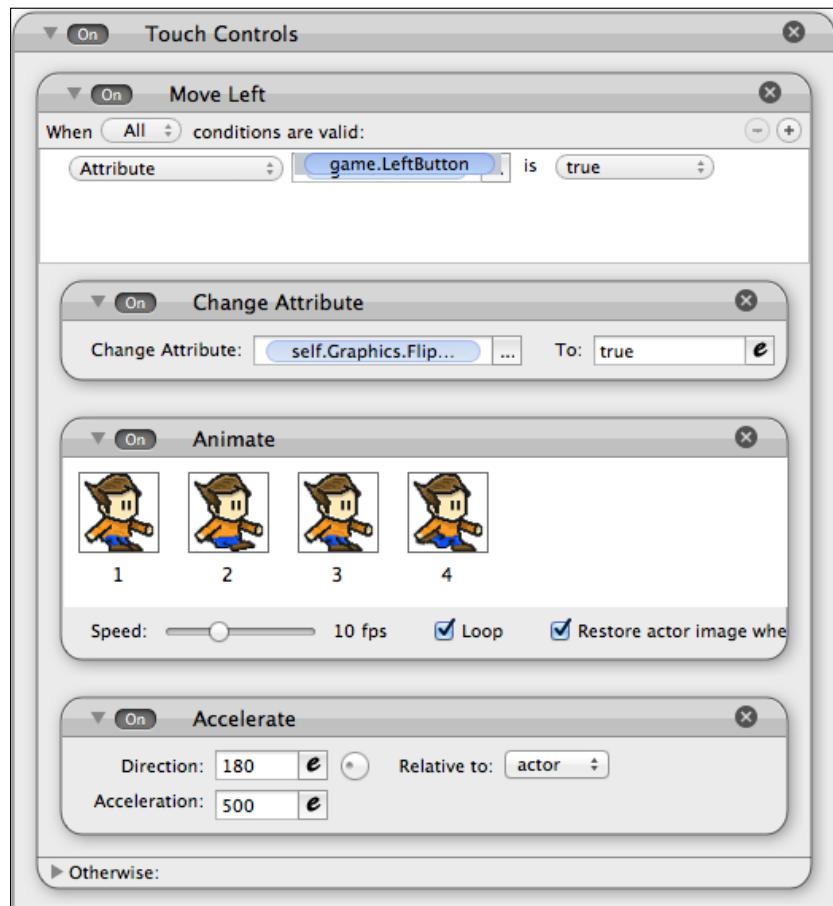
---

What this does is, any time the player isn't touching that button, the `LeftButton` attribute is `false`. Easy! Do the same for each of the other buttons, but change each attribute accordingly. If you want, you can even change the color or image of the actor when it's pressed so you can actually see what button you are pressing.



Simply adding three Change Attribute behaviors to the rule and changing the colors to 1 when being touched, and 0 when not, will highlight the button on touch. It's not required, but hey, it looks a lot better! Let's go back into our Kevin actor, copy and paste the **Keyboard Controls** group, and rename it to **Touch Controls**. Now we have to change each rule from **Actor receives event | key**, to **Attribute | game. LeftButton | is true** (or whatever button is being pressed). Test it out to see if everything works. If it does, awesome! If not, go back and make sure everything was typed in correctly.

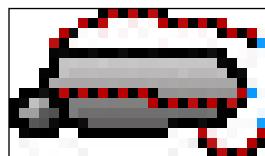
[  Don't forget, when it comes to changing attributes, you can't simply type in game.LeftButton, you have to click on the ... button beside the **Attribute**, and then click on **Game**, then click **LeftButton**. ]



Now that Kevin is alive, let's make him armed and dangerous. Go back to the **Scene** button within the scene editor, and under the **Layers** tab click on the **Background** layer to start creating actors in that layer. If you forget to do this, GameSalad will automatically start creating actors in the last selected layer, which can be a fast way of creating objects, but annoying if you forget.

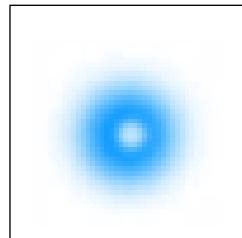
## Attacking

Pretty much any game out there nowadays has some kind of shooting or attacking in it. Whether you're playing **NHL**, **NFL**, or **Portal 2**, there is some sort of attack, or shot that you can make, and Kevin will locate a special weapon in the first level.



The Inter Dimensional Gravity Disruptor

I created both, the weapon and the projectile that will be launched from the **Inter Dimensional Gravity Disruptor**. For the gun actor, don't forget to uncheck the **Movable** option in the **Physics** drop down to prevent the gun from moving around.



Now let's place the gun somewhere near the end of the level. For now, let's create a new game-wide attribute, so within the scene, click on the **Game** button within the **Inspector** window, and open the **Attributes** tab. Now, click on the **+** button to add a new attribute. It's going to be a **Boolean**, and we will name it `GunCollected`. What we are going to do is, when Kevin collides with the collectable gun at the end of the level, the Boolean will switch to `true` to show it's been collected, then we will do some trickery that will allow us to use multiple weapons.

We are going to add two more new game-wide attributes, both Integers; name one `KevX`, and the other `KevY`. I'll explain this in a minute.

Open up the Kevin actor in the **Inspector** window, create a new **Rule**, and change it to: **Actor receives event | overlaps or collides | with | actor of type | Inter-Dimensional-Gravity-Disruptor**, or whatever you want to name it; then drag in a Change Attribute behavior into the rule, and change it to `game.GunCollected | to true`. Finally, outside of this rule, drag in two **Constrain Attribute** behaviors and change them to the following:

```
game.KevX | to | self.position.x  
game.KevY | to | self.position.y
```

The Constrain attributes do exactly what it sounds like, it constantly changes the attribute, constraining it; whereas the Change Attribute does it once.

That's all we need to do within our Kevin actor. Now let's go back to our gun actor inside the **Inspector** and create a new **Rule**, change it to **Attribute | game.GunCollected | is true**.

Inside that rule, drag in two Constrain Attribute behaviors, and change them to the following:

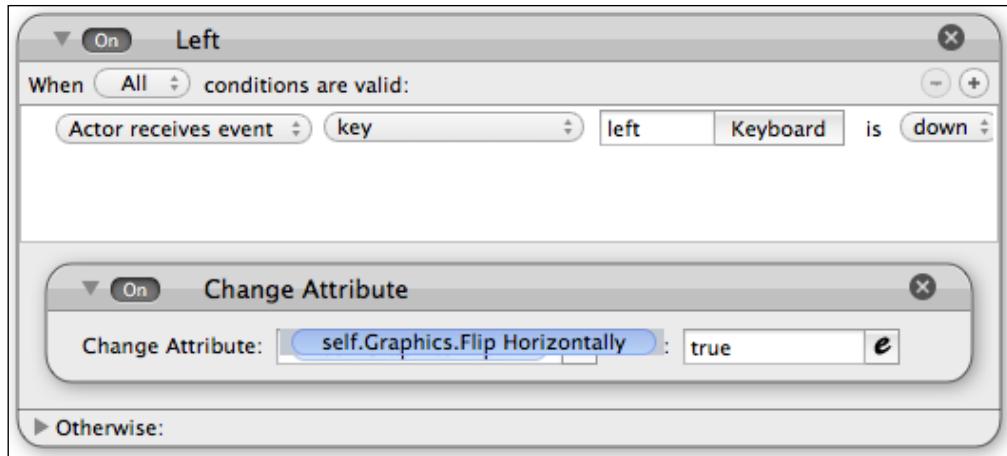
```
self.position.x | to | game.KevX  
self.position.y | to | game.KevY +2
```

Test out your game now and see if the gun follows Kevin when he collects it. If we were to put a Change Attribute instead of Constrict Attribute, the gun would quickly pop to his position, and that's it. You could fix this by putting the Change attributes inside of a Timer behavior where every 0.0001 second it changes the attributes, but when you test it, the gun will be following quite choppily and that doesn't look good.

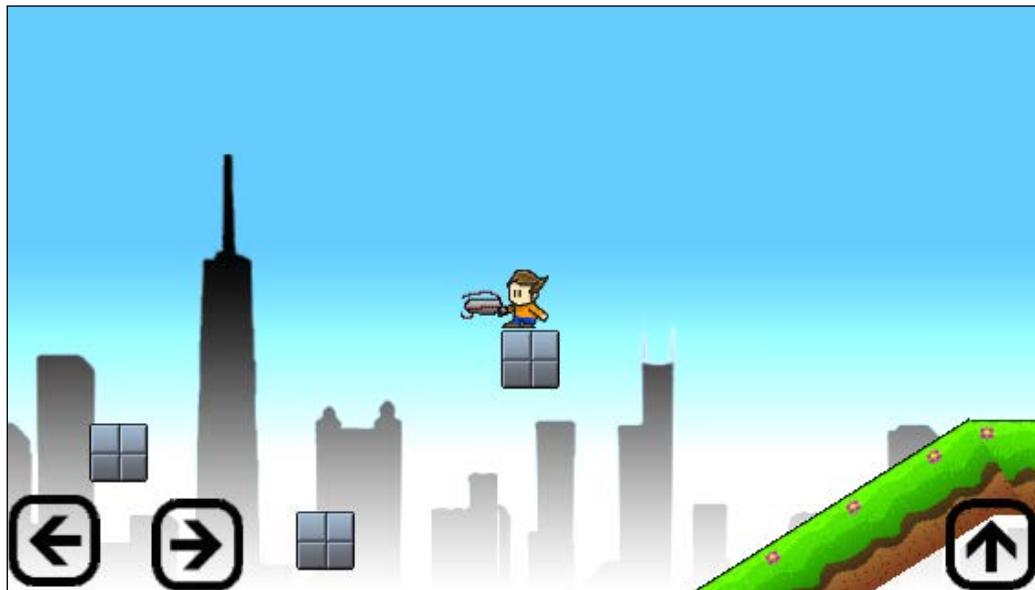
We now want the gun to flip at the same time Kevin does, so let's create two rules the same way we did with Kevin. When the Left arrow key is pressed, change the `flipped` attribute to `true`, and when the Right arrow key is pressed change the `flipped` attribute to `false`. This happens only when the **GunCollected** attribute is **true**, otherwise the gun will be flipping before you even collect it. So create these rules within the **GunCollected is true** rule.

## Diving In - Player Creation

---



Now the gun will flip with Kevin! Keep in mind, if you haven't checked out the gun image I created (I drew the gun on top of the Kevin sprite) and saved it where it was positioned, no centering will happen. This way the gun is automatically positioned and you don't have to finagle positioning it in the behaviors.

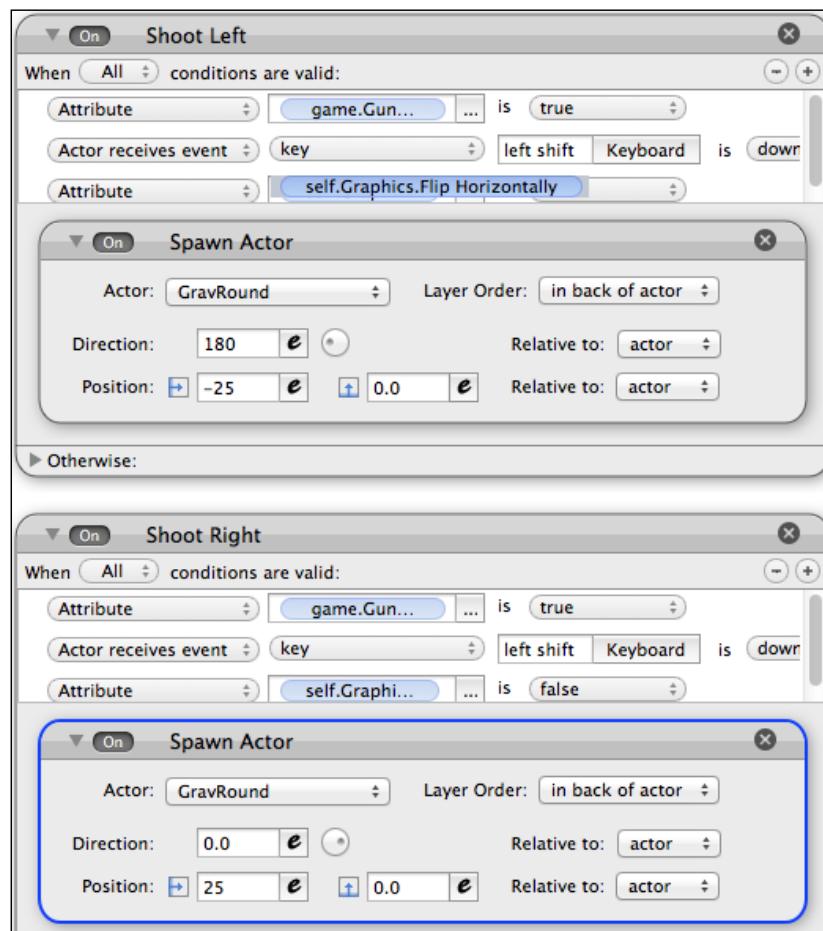


Now, for the pew pew. Open up our **GravRound** actor within the **Inspector** window, simply add in a Move behavior and make it pretty fast; I did 500. Change the **Density**, **Friction**, and **Bounciness**, all to zero.

Open up our **Gun** actor. We are going to create a new **Rule**; it will have three conditions to the rule (for **Shoot Left**) and they are as follows:

- **Attribute | game.GunCollected | is true**
- **Actor receives event | key | (whatever shoot button you want) | is down**
- **Attribute | self.Graphic.FlipHorizontally | is true**

Now drag in a Spawn Actor behavior into this rule, select the **GravRound** actor, **Direction: 180**, **Position: -25**. Copy and paste this rule and change it so flip is false, and the **Direction** of the Spawn Actor is **0.0**, and **Position** is **25**:



Test it out to make sure he's shooting in the right directions. When developing for a while, it's easy to get tired and miss things.

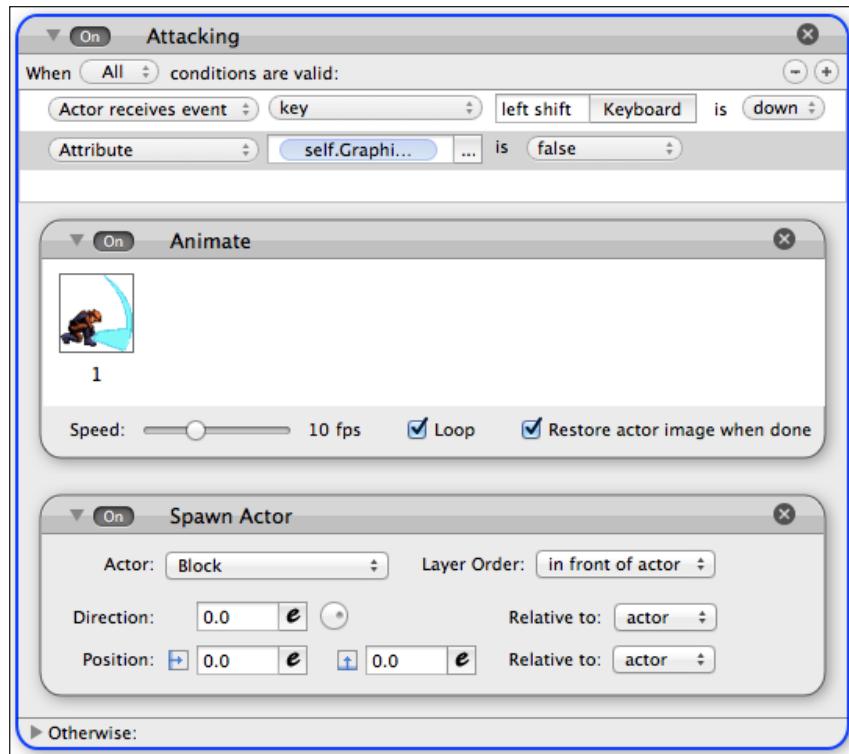
Don't forget to create a new button for shooting, if you are creating a mobile version of the game. Shooting with touch controls is done the exact same way as walking and jumping with touch buttons.

## Melee weapons

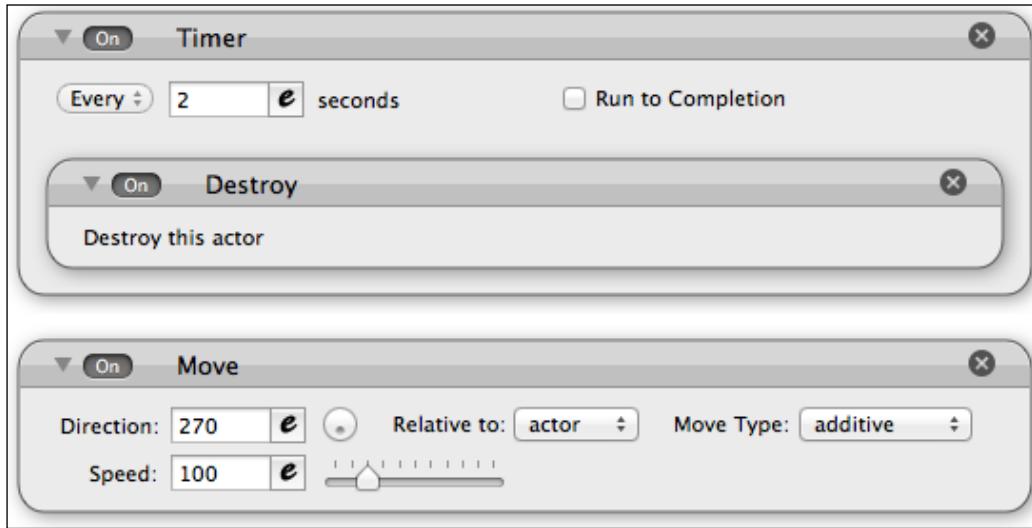
If you want to create a game with melee weapons, I'll quickly take you through the process of doing it. Melee weapons can be made the same way as shooting a bullet in real. You can either have the blade or object as a separate actor positioned to the actor, and when the player presses the attack button, have the player and the melee weapon animate at the same time.

This way when the melee weapon collides with the enemy, it will be more accurate. If you want to have the player draw the weapon in hand, simply animate the actor, and detect if there is a collision.

This will be a little less accurate, as an enemy could run into the player when he is animating and get hurt, so what you can do is have the player spawn an invisible actor to detect the melee weapon collisions more accurately. I'll quickly go through this with you.



When the player presses the attack button and the actor isn't flipped (so he'll be facing the right end of the screen), animate accordingly, and then spawn the actor in front of the player.



Then, in the detector actor you spawn when attacking, have it automatically destroyed when the player has finished animating.

Also, throw in a Move behavior, especially when there is a down swipe such as the attack. For this example, you'll want a detector to follow the edge of the blade. This way, all you have to do is detect collisions inside the enemies with the detector actor.

It is way more accurate to do things this way; in fact you can even do collisions with walls. Let's say the actor punches a wall and this detector collides with the block, spawn some particles for debris, and change the image of the block into a cracked image.

There are a lot of cool things you can do with melee weapons and with ranged weapons. You can even have the wall pieces have a sort of health bar, which is really a set amount of times a bullet or sword will have to hit it before it is destroyed.

That's all we are going to do in this chapter, but hey, we accomplished a lot! Kevin is now bouncing around our level, running like a silly little boy, and now he can shoot things up.

## **Summary**

Before we started this chapter, we had an unmovable, boring character – albeit a good looking one. We discussed how to get him moving using a keyboard, and mobile touch controls. We then discussed how to get our character to collect a weapon, have that weapon follow him around, and then start shooting it.

Sit tight for the next chapter when we dive even further into developing this game! What are we going to talk about, you ask? Well, we are going to make Kevin have health, lives, and power ups. We'll discuss an inventory system, scoring, and some more game play mechanics like pausing and such.

Relax for a little bit, take a nice break. If the weather is as nice out there as it is at the time of me writing, go enjoy the lovely sun. I can't stress the importance of taking a break enough when it comes to game development. There have been many times when I've been programming for hours on end, and I end up making a mistake and because I'm so tired, I can't find the problem. As soon as I take rest, the problem is easy to fix and I'm more alert.

# 5

## Diving In – Gameplay Mechanics

So far you have done some cool new things with our game. Kevin now moves around, you've made menus and cut scenes. Things are looking pretty awesome! Now, because this game is somewhat a **platformer/RPG**, there are a few more mechanics that we should add in. Here is what we are going to discuss in this chapter:

- Creating health, dying, and adding lives
- Creating a scoring system
- Creating great power up, ammo, and inventory systems
- Working a little more on GUI (Graphic User Interface) elements

We are going to have a lot of fun in this chapter! Let's just dive right into it, shall we?

## Health, dying, and lives

"No Mr. Kevin, I expect you to die." Unfortunately for Kevin, by the end of this chapter, he will die. Poor Kevin! All he wants to do is save Dom. No pain, no game! Let's get into it. Let's open up a level; we are going to create two new game-wide attributes. Don't remember how? No problem. There's a lot of information to handle here! In the scene editor, and in the **Inspector** window, make sure the **Game** button is clicked, then click on the **Attributes** tab, and then click on the **+** button to add a new attribute. Both of our new attributes will be Integers; call one **Health** and the other **Shields**. Change the **Health** value to 100, and leave **Shields** at 0. We are going to create a power up that will give Kevin shields. We are now going to create two new blank actors. Again, name one of them **Health Bar** and the other **Shield Bar**. Edit each one; change the **Width** to 100, the **Height** to 2, the Color for the health bar to red, and for the shield bar blue. In the health bar, we are going to add in two **Constrain Attribute** behaviors; we are going to change the first attribute's values to **self.Position.X To: game.KevX**, and the second one to **self.Position.Y To: game.KevY + 25**, which means the bar will be floating above Kevin's head. Do the same for the shield bar, but instead of placing it 25 pixels above his head, do 30.

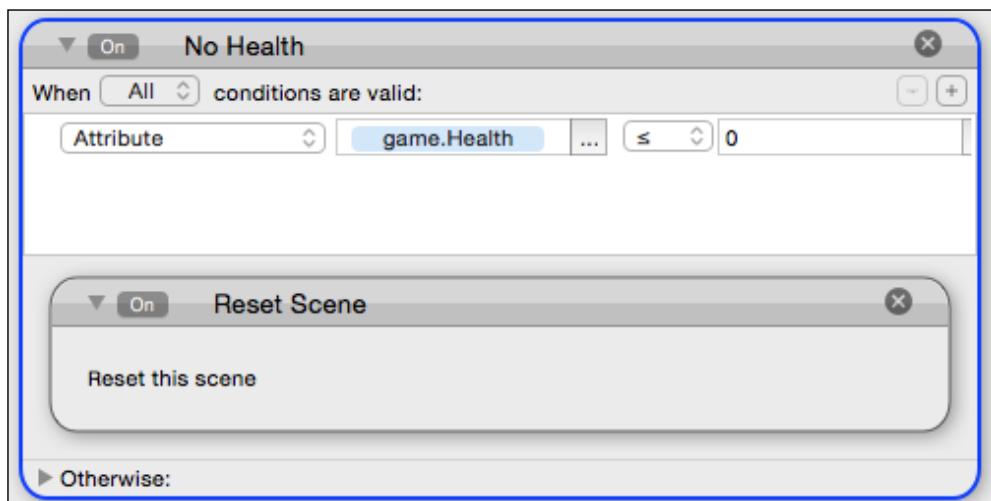


Finally, for the health bar, we are going to add in a **Constrain Attribute** behavior, and change it to the following:

**Constrain Attribute:** `self.Size.Width to: game.Health`

Do the same for the shield bar, but instead of `game.Health`, change it to `game.Shields`.

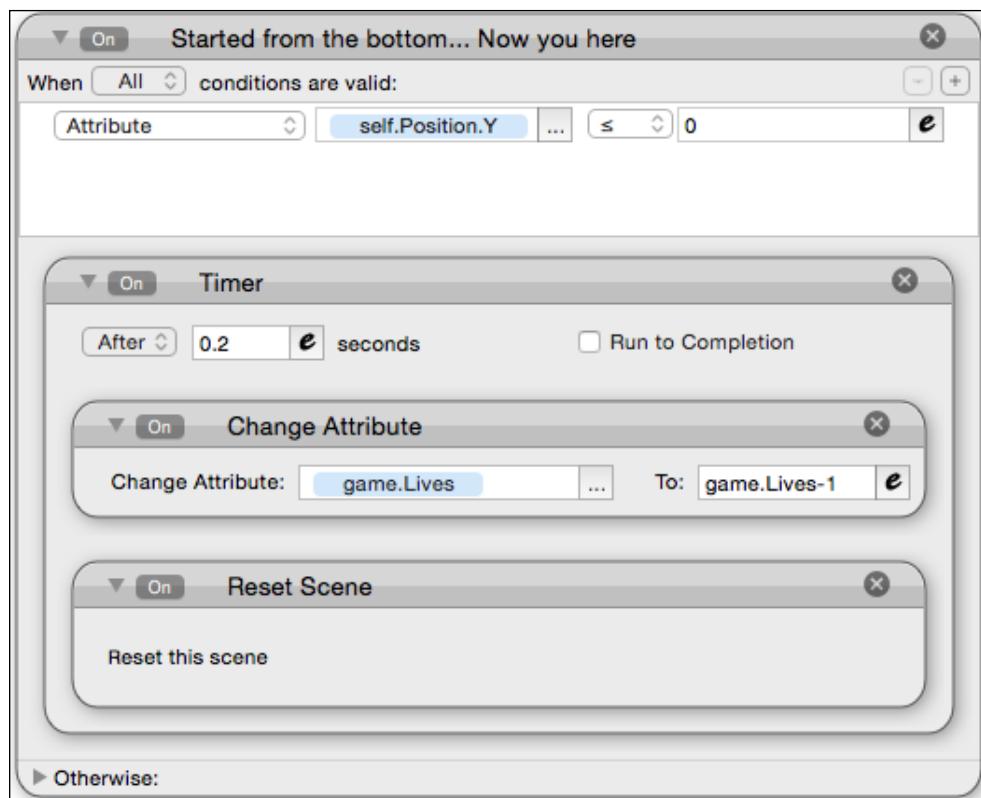
Let's now edit Kevin. Create a new group and call it `Health`. We are going to keep all our health-related behaviors and rules inside this group to keep everything organized. First off, we are going to spawn the two bars; so drag in two `Spawn Actor` behaviors and select both the bars accordingly. You don't have to worry about positioning because the bars are already constrained to the correct positions. Drag in a `Change Attribute` behavior, and change it to `Change Attribute: game.Health to: 100`. This way, whenever a new level is started, he has fresh full health. All we have to do when it comes to dying is create a new rule, change it to `Attribute: game.Health < 0` (which means when the player runs out of health, or if they reach below 0, which can happen, let's say you have 5 health remaining, and an enemy hits you with 25 damage. We need to add that variable in as well), then drag in a `Reset Scene` behavior. It is that simple. You could also do a dying animation, and when the animation has completed, reset the scene.



We also have cracks and holes our hero can fall through. Instead of creating a new actor at the bottom of the cracks and holes, we will just detect if he has gone below the bottom of the screen and we will reset the scene.

Create a new **Rule**, change it to `Attribute: self.Position.Y < 0` (when the player's position has reached 0, or the edge of the screen, or lower), then drag in a `Reset Scene` behavior; or you can have a `Timer` behavior and after a certain amount of time you can reset the scene so it's not as abrupt, or so you have time to play a quick sound effect.

Now all you have to do is create a new rule. When Kevin collides with an enemy, use **Change Attribute: game.Health to: game.Health -** (whatever value you want enemies to take off). Now let's add in some lives. To do this, all we have to do is create a new game-wide attribute (Integer of course), and name it **Lives**. Change the value to 5. If you want, you can also have a selection on the main menu to select your difficulty, and depending on what difficulty you select, you can adjust health and lives. Then add in your **No Health**, and **Bottom of the Screen** rules before the **Reset Scene** behavior. Add in a Change Attribute behavior and change it to **Change Attribute: game.Lives to: game.Lives - 1**:



Now you can add in a rule inside the Kevin actor detecting when the `game.Lives = 0`, have it change the scene to a game over screen. From there, you can have a button to continue or go back to the main menu. Once one of the buttons has been pressed, reset the lives back to 5 or the number you chose.

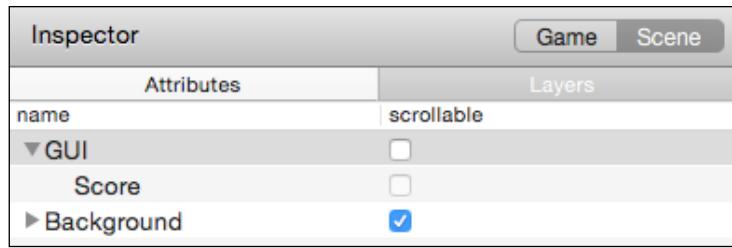
That's all for health, dying, and lives! It's super simple. Now let's talk about scoring, and continuing our game.

## Scoring and Saving Games

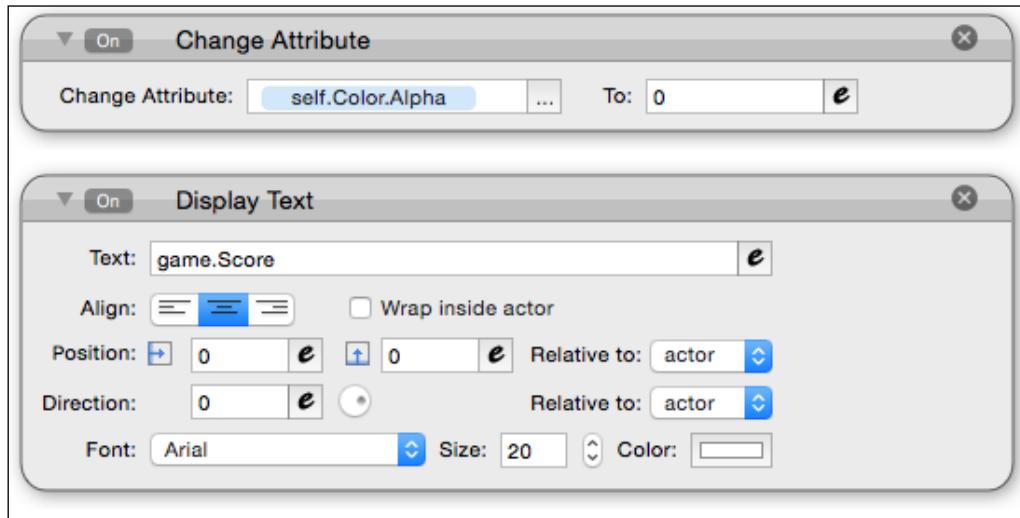
Remember in an earlier chapter we talked about saving the game? I bet you thought I forgot about that, didn't you? I certainly didn't! We will get to that in a moment, but first we are going to set up the scoring system.

In the scene editor, create a new game-wide attribute; it's going to be an Integer. Name it `Score`. Now create a new actor, name it `Score`, or something that you'll remember. Open it up so we can edit it. Change the Size to something small, like 32 x 32. Drag in a **Change Attribute** behavior, and change it to **Change Attribute: self.Color.Alpha to: 0**. This will hide the actor only when you hit play, so when you are placing it within the scene, you will see it.

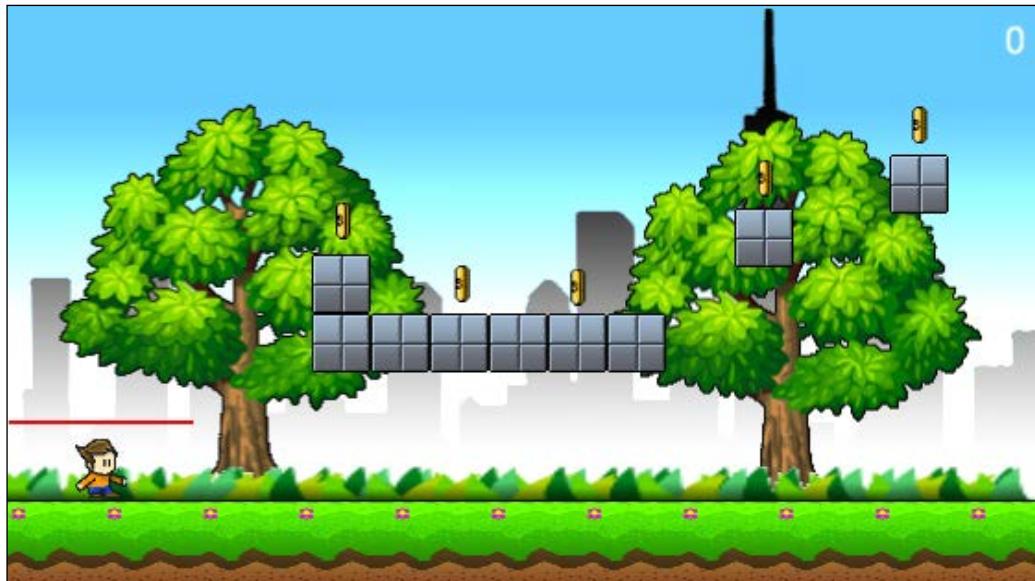
Now drag in a **Display Text** behavior, click on the `e` button to open the expression editor, and change it to `game.Score`, so it displays the score value:



Finally, don't forget to uncheck the **Movable** option under the **Physics** drop down to prevent it from falling off the screen, and move it into a non-scrollable layer so it follows the screen when you start walking.



Now let's create some coins throughout the level that the player can collect. We can create a marketplace at the end of each level where you can spend your points.



We will now edit Kevin. Let's create a new Rule, and change it to **Actor receives event | overlaps or collides | with actor of type | coin**, then drag in a Change Attribute and change it to **Change Attribute: game.Score | to: | game.Score + 10**. This way, whenever Kevin collides with a coin, it will add to the score. Now we have to edit the **Coin** actor. Simply create a new rule, change it to **Actor receives event | overlaps or collides | with actor of type | Kevin**, then drag in a **Destroy** behavior. If you want to, you can also make a quick animation, like the coin glittering when it disappears (like in Sonic).

Now that we have the scoring completed for the moment, let's work on saving the level we are on, and let's save the score as well.

There are two different ways we can save attributes. One way is by simply using the **Save Attribute** behavior. However, after some research, it does not seem to be the most reliable way to save information.

So we are going to use tables. Go into the **Tables** section and create a new table by clicking on the **+** button, name it `Saved Data`. Change it to two rows and one column. Change the first row header to an **Integer**, and rename the first cell `Score` and the second one `Level`.

Leave all the values at `0` for now, we will write the tables during gameplay. Now let's go back to our first level and, at the very end, create a goal for Kevin to reach. This can be an actor off the edge of the screen, or it can be a castle, like in Super Mario.

Saved Data			Rows <b>2</b>	Columns <b>1</b>
			1	
			Integer	▼
1	<b>Score</b>	0		
2	<b>Level</b>	0		

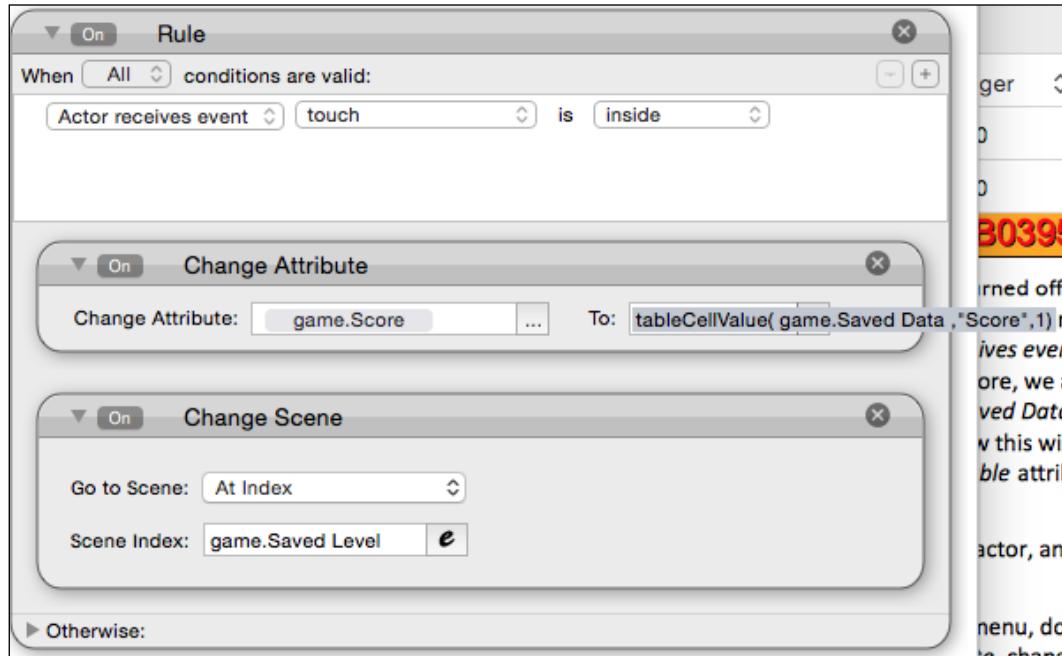
Once you have your goal placed and you've turned off the **Movable** option in the **Physics** drop down so it's not bouncing around or falling through the floor, we are going to edit this actor a bit. First off, let's create a new rule, change it to **Actor receives event | overlaps or collides | with actor of type | Kevin**. First, we are going to save the score. We are then going to add a **Change Table Value** behavior, change the **Table** value to `game.Saved Data`, **Row:** `1`, **Column:** `1`, and **Value:** `game.Score`. Now this will write the current score at the end of each level. Finally, we need to drag in a **Save Table** behavior, and select our `Saved Data` table so we can finalize the writing on the table.

Alternatively, you can do this within the `Score` actor and have it write and save the table every second, or faster.

Now all we have to do is go back to our main menu, double-click on our **Continue** button, add in a **Touch** rule and add in a **Change Attribute**, and change it to **Change Attribute: game.Score | to: | tableCellValue(game.Saved Data, 1, 1)**.

## Diving In - Gameplay Mechanics

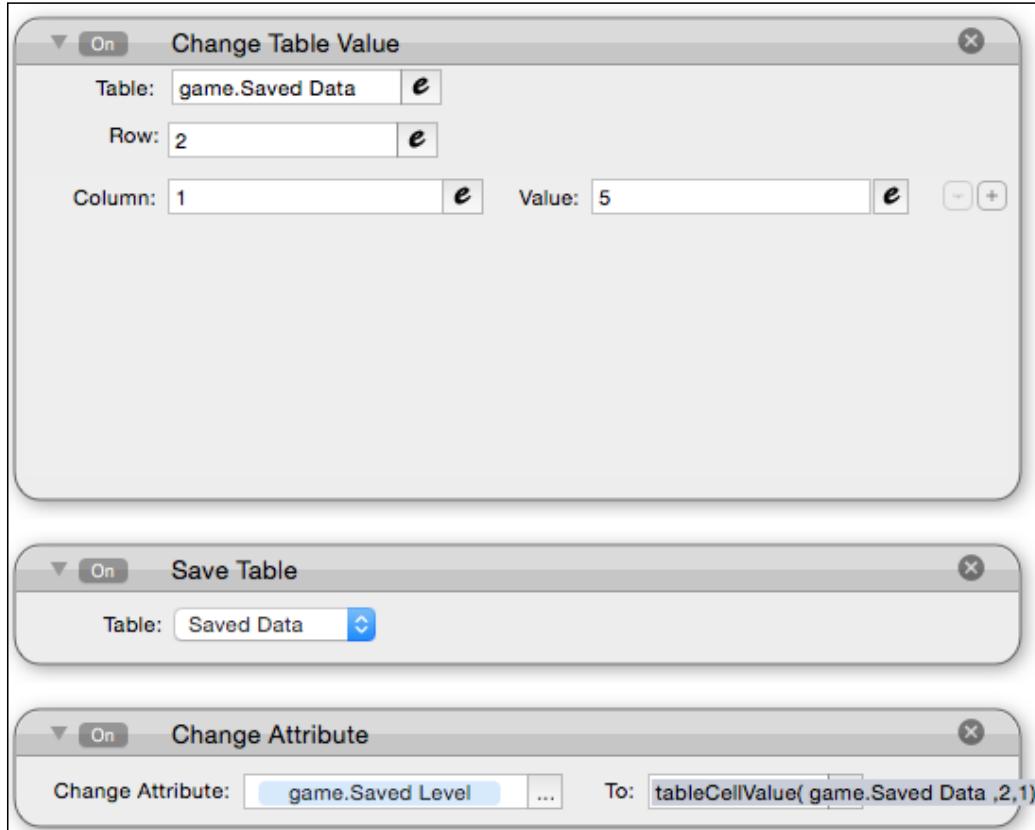
You can also write the name of the row, as opposed to writing 1 or 2. You can write Score, or Level. I tried it; I preferred using Integers instead. But that's just me; you do whatever is easier for you.



When you test your game in the editor, go to the level, collect some coins, and then hop back to the main menu. Click on the **Continue** button and you will start with your saved score. If you run the game again in the editor, it will revert back to 0 because the editor doesn't save the tables. It goes back to what's typed in. When you publish your game, tables will get written and saved properly, according to the GameSalad forums (<http://forums.gamesalad.com>).

Now for saving our levels, everything is essentially the same, but in our **End of Level Goal** actor that's within the scene, you will want to double-click on it, and click on the lock button because the level values are going to be different per level. Simply drag in a Change Table Value behavior, change the **Table:** game.Saved Data, **Row:** 2, **Column:** 1, **Value:** 5 (or according to the scene you are working on), then drag in a Save Table behavior, and select the Saved Data table.

Finally, add in a Change Attribute behavior, and change it to **Change Attribute:** game.Saved Level | **to:** | tableCellValue( game.Saved Data ,2,1). So, now this will change the table, save it, then change the Saved Level attribute to the table value:



Now we have to go back to our **Continue** button in the main menu, and adjust the Change Scene behavior. Instead of game.Saved Level, we will change it to tableCellValue( game.Saved Data , "Level" , 1). I found that when it was loading from the Saved Level attribute, it wouldn't switch to the proper level due to the fact that I didn't create a behavior to change the Saved Level attribute to the table when the game starts. It's easier just to load it directly from the table.

Now we move on to some cool stuff!

## Power ups, ammo, and inventory

The weapon we gave Kevin is a great gun, but there are always bigger and better ones out there! Plus, we are going to make Kevin darn powerful with some cool power ups because he's really going to need them in his journey. By now, things are simple enough and you should be able to create things with no problem, but in case you don't, let's do a couple power ups for Kevin. One will increase his armor and the second will make him invisible for ten seconds.

Let's start with the armor power up. I have created a simple image for the power up. Import it and create a new actor from it. Drag it somewhere into your scene, preferably somewhere out of the way to make it challenging for the player to reach. Double-click on the actor within the **Inspector** window so we can edit it.

I changed the **Blending** mode under the **Graphics** drop down to **Additive** to make it look like it's glowing. Now all we have to do is create a rule, change it to **Actor receives event | collides or overlaps | with actor of type | Kevin**, then drag in a Destroy behavior into the Rule.

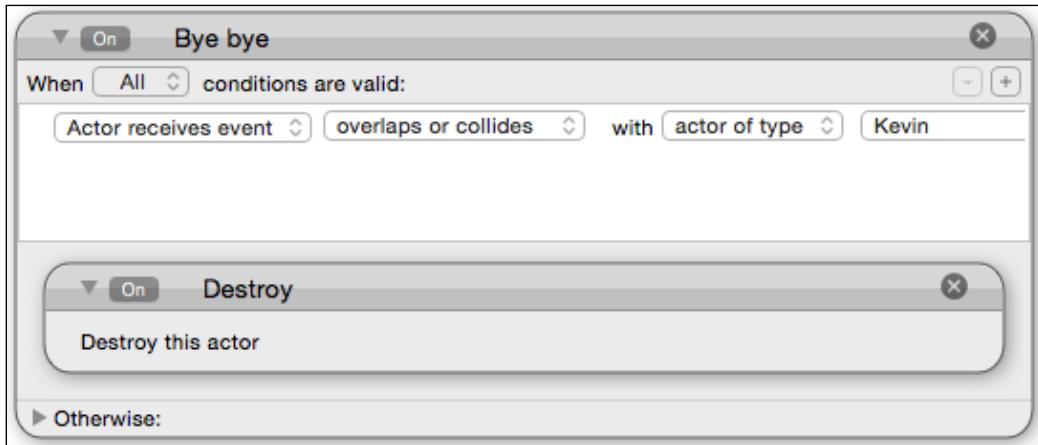
Now in our Kevin actor, we will create a similar rule, but instead, we will have it colliding with the armor power up actor, and then inside the rule, we will drag in a Change Attribute behavior and change it to **Change Attribute: Game.Shields | to: 100**.

Easy! Now Kevin has shields. In the next chapter, when we start creating our baddies, we will discuss his shields a little further, but for now, he has them.

Now for a "Super Mario-esque" star that will keep Kevin invincible for ten seconds, I think it would be cool if you shoot a treasure chest and it spawns the star.

I have also created an image for the star. Import it into GameSalad and create an actor for it. We will import the treasure chest used earlier into the scene. Let's double-click on it to edit it. Create a new rule, and change it to **Actor Receives event | overlaps or collides | with actor of type | GravRound**, then drag in a Spawn Actor behavior and select the **Star** actor, and then drag in a Destroy behavior.

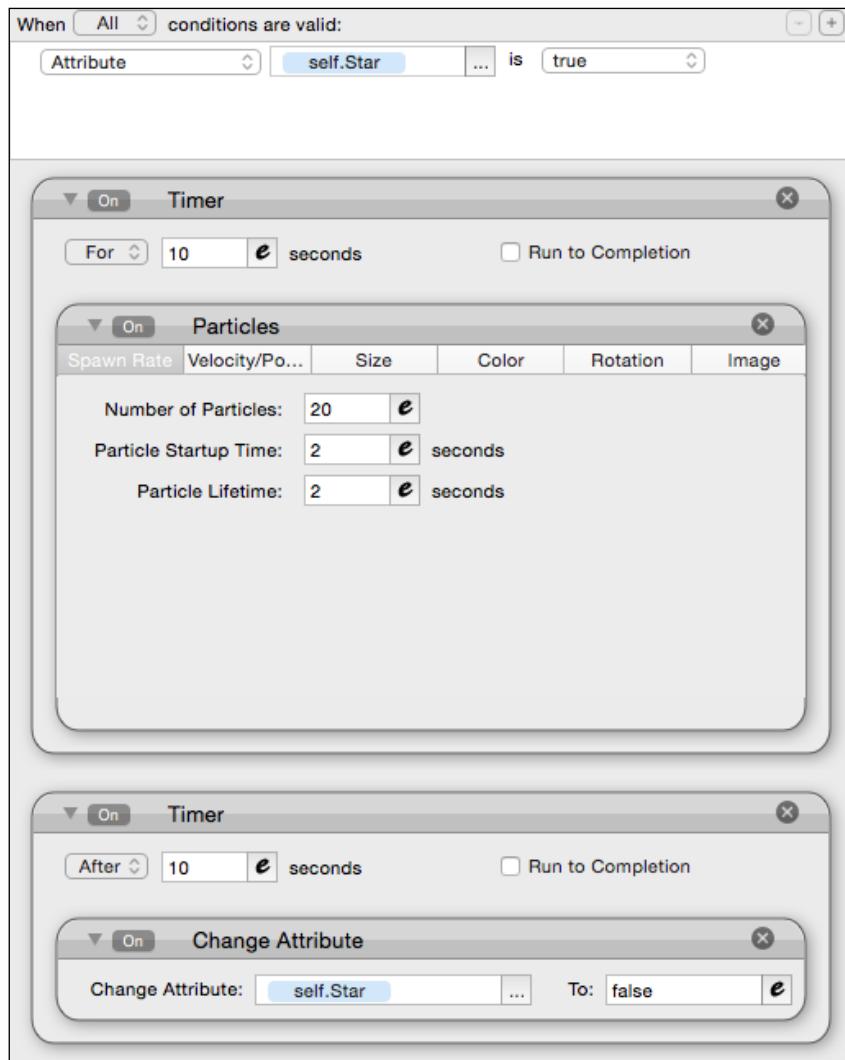
Now let's go into our Star actor, make sure the **Movable** option is not selected. Then create a new rule so that when Kevin collides with the star, this object is destroyed:



Now back into our Kevin actor, we are going to create a new attribute to detect that he has collected the star. Click on the **+** button in the **Attributes** pane of our Kevin actor, make the attribute a **Boolean**, and name it `Star`, or something easy to remember.

Next, create a new **Rule** and change it to **Actor receives event | overlaps or collides | with actor of type | Star**, then drag in a **Change Attribute** behavior into this rule, change it to **Change Attribute: self.Star | to: | true**.

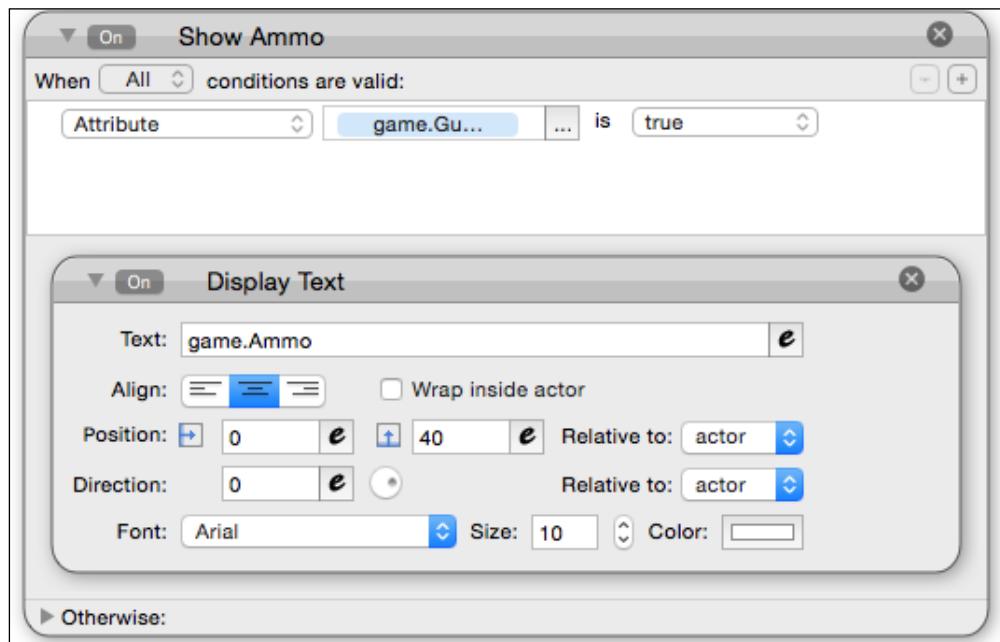
We are now going to create another rule to spawn some particles to make Kevin look all flashy and such. Change the rule to **Attribute | self.Star | is | true**, and drag in a Timer which we will change it to **For 10 seconds**, then drag in a **Particles** behavior. All you have to do is change the image. I selected the star image and it looks pretty cool. Finally, inside the Timer behavior, drag in another Timer (add value **After 10 seconds**). Drag in a Change Attribute behavior inside this Timer and make **Change Attribute: self.Star | to: | False**:



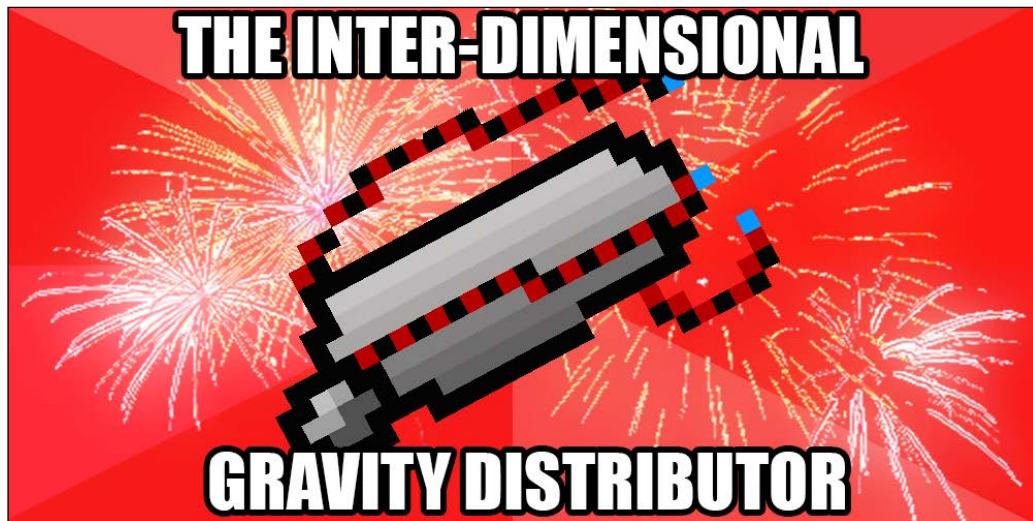
Test it out to see if everything works. If it does, Kevin should emit particles for 10 seconds, then stop. Again, in the next chapter, when we are creating our enemies, we will talk more about Kevin taking damage from them, and how we can differentiate between getting hit normally and when he has a star.

## Ammo

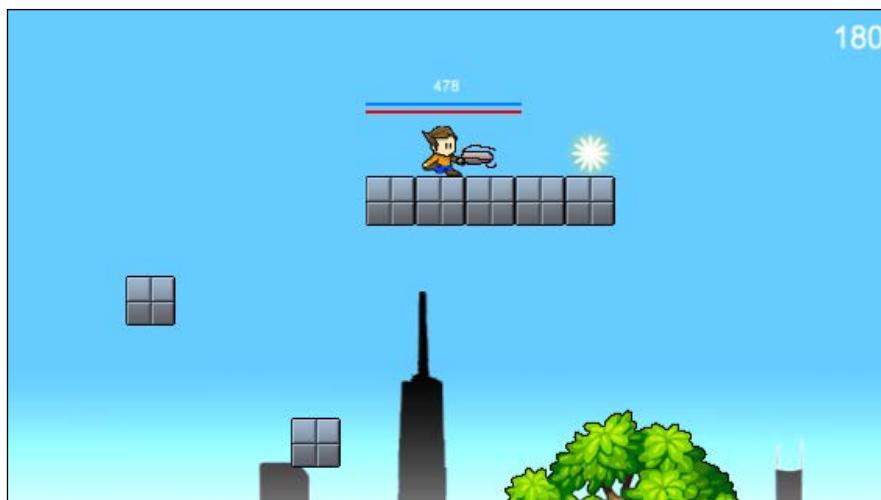
While we could let Kevin have an unlimited amount of ammo, what kind of challenge would that be? Ammo is another easy thing to program into GameSalad. We will start by creating a game-wide attribute, which will be an Integer, and we will call it `Ammo`, change the **Value** to 500. Alternatively, you could set it to a low number such as 7, and create another attribute for the amount of clips; then once you run out of ammo, decrease the number of clips and reload the ammo. However you want to do it, I'm going to show you how to do it! In our Kevin actor, create a new rule, change it to **Attribute** | `game.GunCollected` | **is** | `true`, and then drag in a Display Text behavior. For the text click on the `e` button to open up the expression editor and select `game.Ammo`, and set the **Y** position to 40, and the **Size** to 10. Once you collect the gun, it will display the ammo value above Kevin's head. Handy!



To count ammo that is being depleted, go into our Inter-Dimensional Gravity Distributor actor. That didn't sound epic enough. Hold on!



That's better! Let's go edit our weapon, and inside both the **Shoot Left** and **Shoot Right** rules, we need to add in one more condition each. In the rule, click on the **+** button, change the new condition to **Attribute | game.Ammo | > | 0**. This way, he will only be able to shoot when he has more than one bullet. Now in both shooting rules, drag in a Change Attribute behavior, and change it to **Change Attribute | game.Ammo | to | game.Ammo -1**. This will now deplete the ammo each time you shoot.

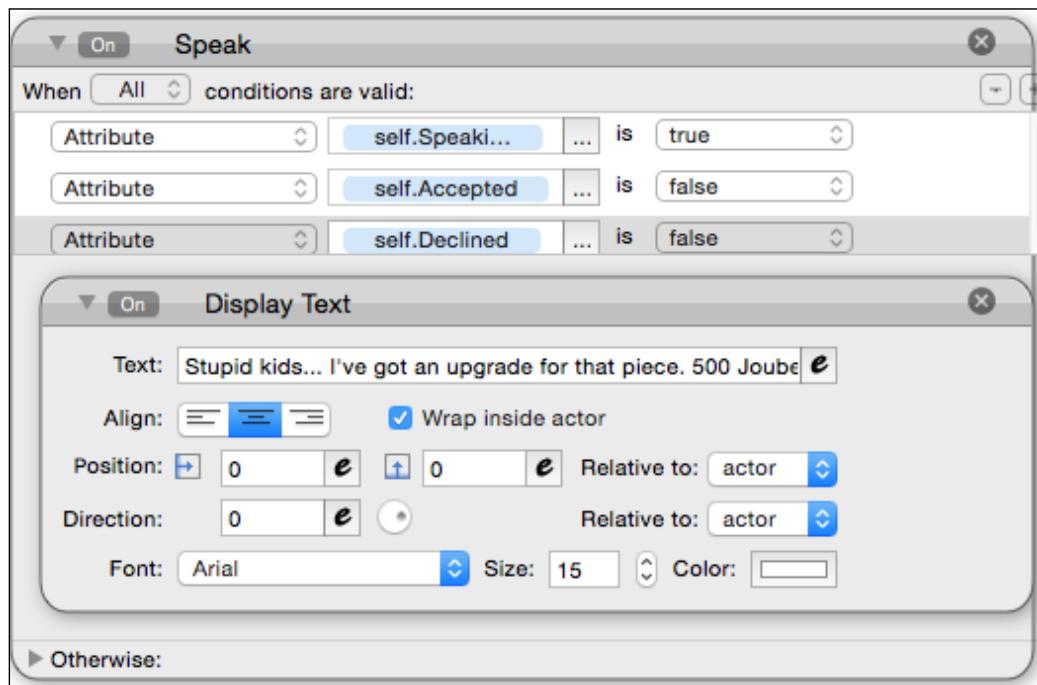


Now all you have to do is create an ammo power up so that when Kevin collides with it, it will replenish his ammo. Check out the *Power ups, ammo, and inventory* section in this chapter to see how to do it.

Want different guns? Easy! Let me show you how. Let's say after this level you go to a town and in that town, you will have a merchant who will sell you supplies. I created a new level, populated it a bit, and created a merchant actor. The merchant will have three new Booleans; one for his speech so the conversation only comes up once after which the player will have to press a button to interact with him, another for **Accepted**, and the last for **Declined**.

First, we will create a new **Rule, Actor receives event | overlaps or collides | with actor of type | Kevin**, then drag in a Change Attribute behavior, and it will be **Change Attribute: self.Speaking | to | true**.

Create another rule, **Attribute | self.Speaking | is | true**, with two more conditions; **self.Accepted** is **False**, and **self.Declined** is **False**. Then drag in a Display Text behavior and type in something like, "Hey I've got a nice upgrade for that piece, only 500 coins. Press Y to buy or N to decline."

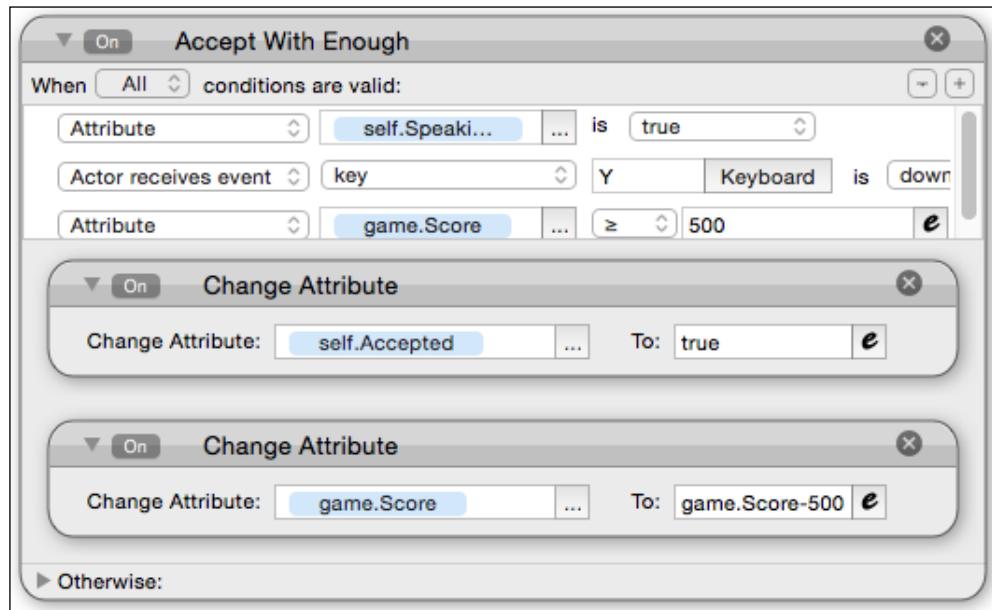


Create a similar rule with two different conditions:

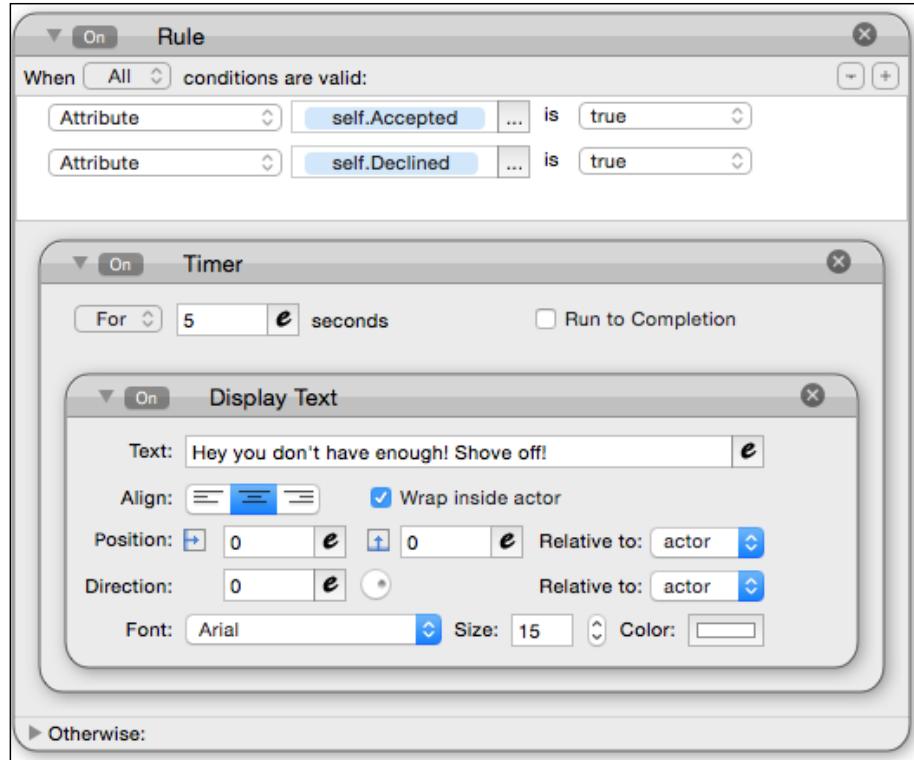
- The first being **Actor receives event | key | Y | is down**
- The second being **Attribute | game.Score > 500**

This will detect if the player presses Y to buy, and has enough coins.

Then debit the player's score by dragging in a Change Attribute behavior, **Change Attribute: | game.Score | To: | game.Score - 500**, and change the Accepted attribute to **True**:



You can add a Timer behavior that will display a message from the merchant. Basically, create a new rule. Let's say if the player accepted it but didn't have enough money, turn on both, Accepted and Declined, then in the **Message** rule, check whether they are both **True**, then add a Timer for 5 or 10 seconds to display the message:



See, there's a lot of flexibility here, so many possibilities. It's like choosing your own adventure novel, except you are actually controlling the outcome!

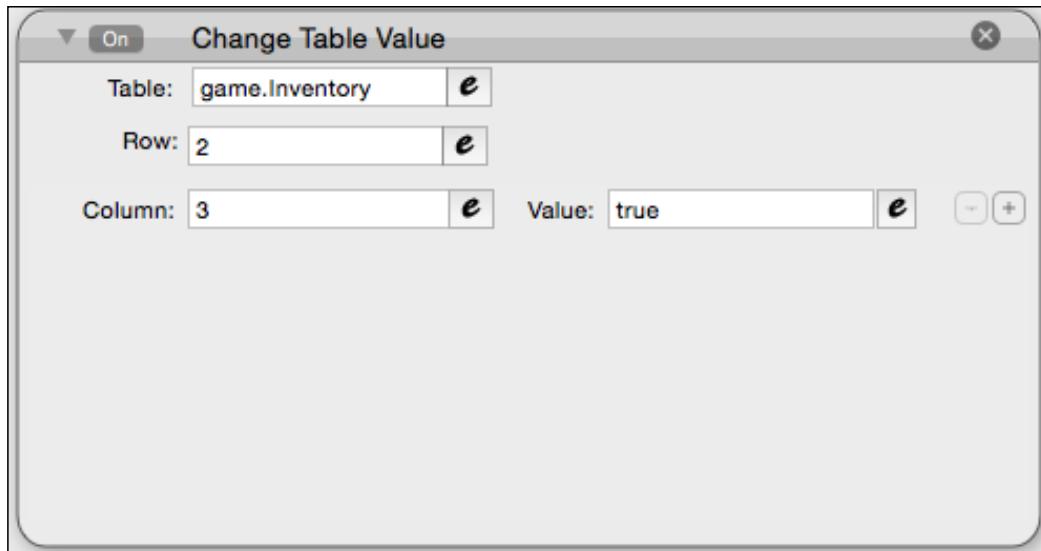


Now let's say you purchased it. What then? Let's go home and create a new table, name it `Inventory`. I did three rows, and three columns:

- The first column I made is **Text**, which will be the name of the item or weapon
- The second one is an **Integer**, which will be the damage it inflicts
- Finally, the third is a **Boolean**, which will show if you own it or not

	1	2	3
	Gun Name	Damage	Owned
	Text	Integer	Boolean
1	GravGun	10	<input checked="" type="checkbox"/> True
2	FlameThrower	25	<input type="checkbox"/> False
3	The Judge	50	<input type="checkbox"/> False

Let's go back to our merchant scene. In the rule where the player accepted and had enough coins, drag in a Change Table Value behavior. The way I had it set up, he would be getting a flame thrower, so that's row two; and we want to turn it on so the column would be three, and obviously the value would be changed into **True**. Then drag in a Save Table behavior, and select the `Inventory` table:

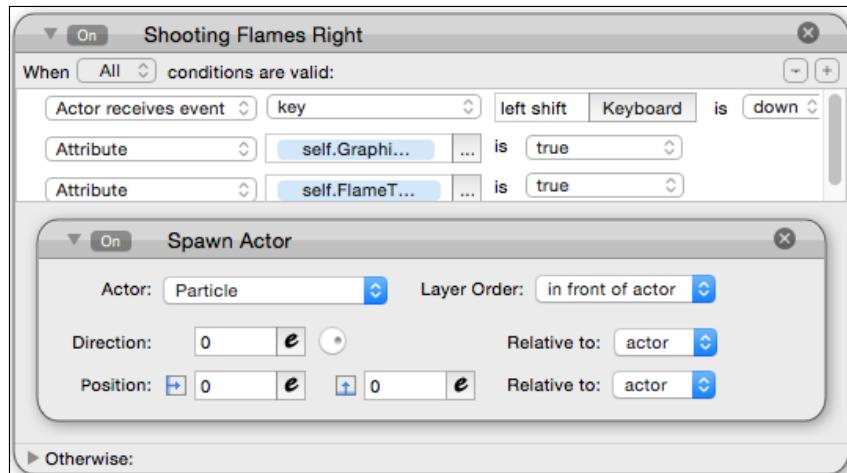


Now we want to display that Kevin now has that weapon, and how to switch to it. We need to create two game-wide Boolean attributes for each of the weapons or items we want in the game—an unfortunate work-around, because as of the time of writing you cannot use a rule to check conditions of a table. Oh well, we aren't going to create one for the gravity gun because we already created one that detects it was collected! So there's one down. For now, I created two new Boolean attributes, and named them accordingly. For the flame thrower inside the merchant actor, under the **accepted with enough coins** option, we need to change the **Flame Thrower** attribute to **True**, so just drag in a Change Attribute behavior and edit it accordingly:

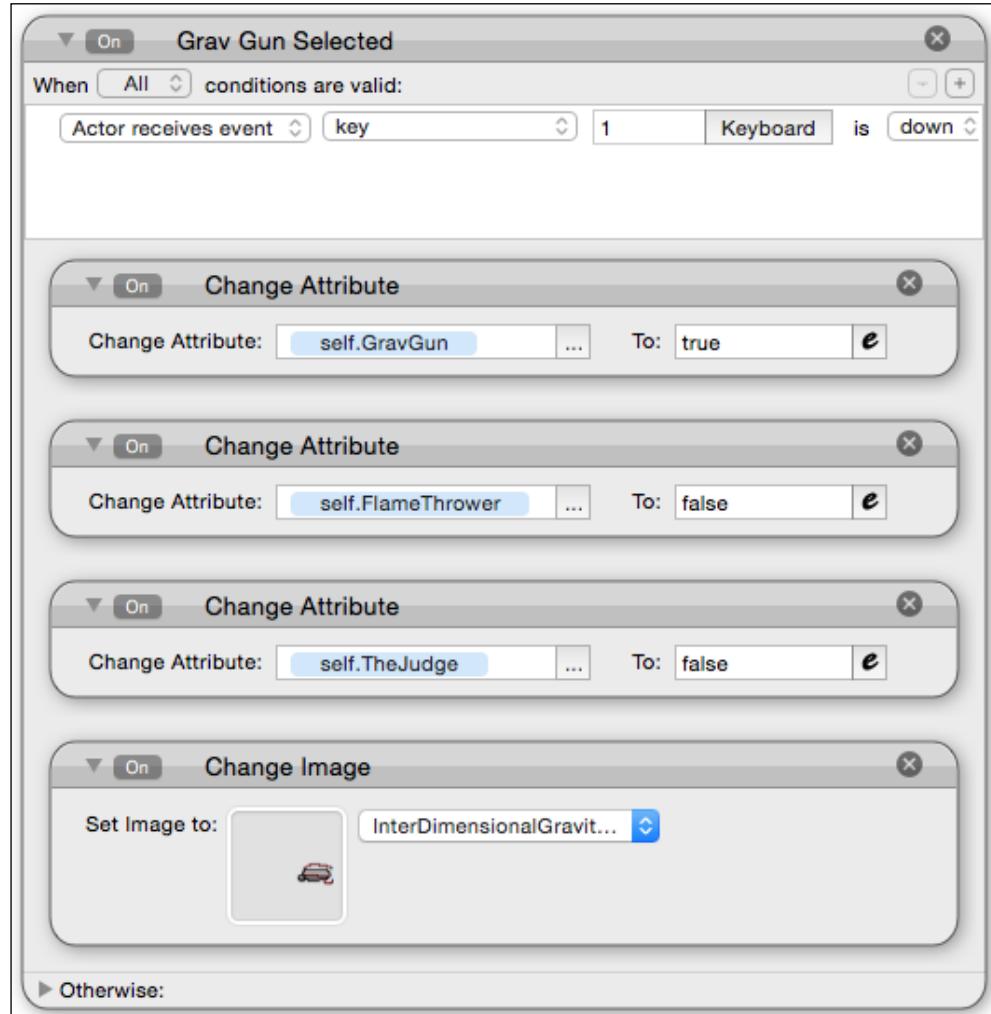
FlameThrower	<input type="checkbox"/>	boolean
TheJudge	<input type="checkbox"/>	boolean

Now that we have turned the attribute on, let's go into our weapon actor. We will be re-working almost all of the behaviors and rules within the actor. So I'll just skim quickly over it to let you know what needs to be changed.

You will need to create three new Booleans within the weapon actor, or however many weapons you want, including the gravity gun, which will be true from the beginning; name them all accordingly. In each shooting rule, you will need to add another condition to detect which gun is being used. For example, **Attribute | gravgun | is | true**. The great thing about this is you can do a whole bunch of guns within the one actor without having to create multiple actors for each gun. So what you can do is, create a new rule. If the player presses the 1 button in the numeric keypad, it will turn on the gravgun Boolean and switch all the others off, change the image back to gravgun. Then at the time of programming the shooting, you can detect whether, let's say the flame thrower is selected, you can have it detect if the Boolean is on, and when the player is holding the shoot button, have the gun emit flames.



And following is how to switch weapons:



Then what you can do to display which weapon Kevin has in his inventory inside the Kevin actor, is use a Display Text behavior, either under Kevin, or in a heads-up display. Detect which Boolean is turned on, and display text, or even an image.



Of course, all the heads-up display, such as the health bars, ammo, and weapon label, don't have to be stuck to the character. Like I mentioned above, you can create a heads-up display, or simple interface graphics like a boarder, or a bar that you would like everything to be put in. Then instead of having the interface rules inside the player actor, you can have them within the UI elements you create.



As you can see, GameSalad can be super complex and powerful. So many different conditions you can test within a rule which adds a lot of flexibility, which is fantastic. Are you starting to see why so many people are choosing to use GameSalad for their game development? I haven't used GameSalad for a while and I'm starting to love it again!

## **Summary**

Boy! We learned a lot in this chapter now, didn't we? We created a health system along with lives and dying. Poor Kevin! Then we discussed a scoring system, which we can use for purchasing items and weapons further into the game. We also created an awesome inventory system that incorporated GameSalad's powerful tables system for storing table information. Then we added a little more interface elements to add to the visual appeal of the game.

In the next chapter, we are going to work on enemies because what game would be complete without some form of antagonist(s)?

Go take a break, and I'll see you in the next chapter.

# 6

## Diving In – Enemies

Would **Super Mario** be complete without **Goombas**, **Halo** without **Grunts**, or **Portal** without **GLaDOS**? No. No real game is complete without some solid enemies. That's just what we are going to work on in this chapter, because Kevin needs someone to stop him from rescuing Dom, right? Let's see what we are going to talk about.

- Creating various enemies to make things difficult for Kevin
- Making our enemies move, including a path finding system
- How to make our enemies attack Kevin
- How to give our enemies health and make them die too

Let's make things hard for Kevin.

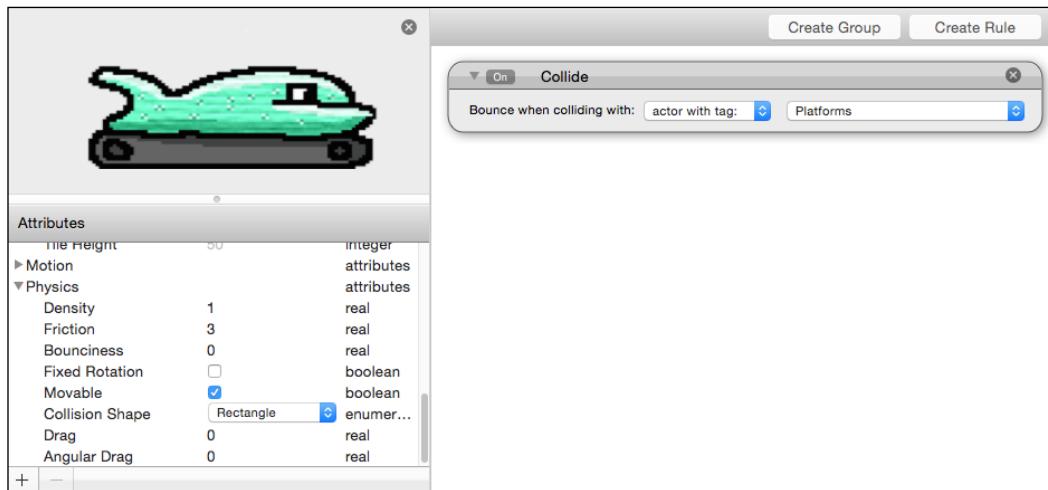
### Creating enemies

Let's start setting up our enemies. If you haven't drawn some up yet, no worries, you can use the ones I've created. Import all the enemy images into our project and create a new actor from the images. Open it up so we can bring the enemy to life.

First thing we should do is change his **Bounciness** to 0; we don't want him bouncing all over the place.

## Diving In - Enemies

Secondly, we want to drag in a **Collide** behavior, and change the options so it's colliding with **Any** actor in the **Platform** tag.



If you want, you can de-select the **Fixed Rotation** option under the **Physics** drop down. This way, if your enemy falls off a platform, he doesn't flip around like crazy and fall flat on his face. For this guy, however, I am going to allow him to automatically rotate because when he reaches inclines and declines, I want him to rotate accordingly.

I also changed his **Max Speed** option to 250 so he doesn't start moving superfast.

 Try this  
Do you want an infinite horde of enemies instead of having to place them all in the level? Why not have an off-screen actor spawning enemies at random intervals and random positions? This is great for top-down platformers, or even racing games to spawn cars on the road!

Now we've created an awesome new enemy! He doesn't do much though. He's kind of boring, isn't he? What would make him even better? Oh, duh! Moving and attacking and all that fun stuff! Let's make him move!

## Motion and path finding

Enemy movement is fairly easy, the same as pretty much everything else in GameSalad. First off, we are going to drag in an **Accelerate** behavior, and set the speed to 250 and its direction. When it comes to directions, 0 moves the actor right, and 180 moves him left.

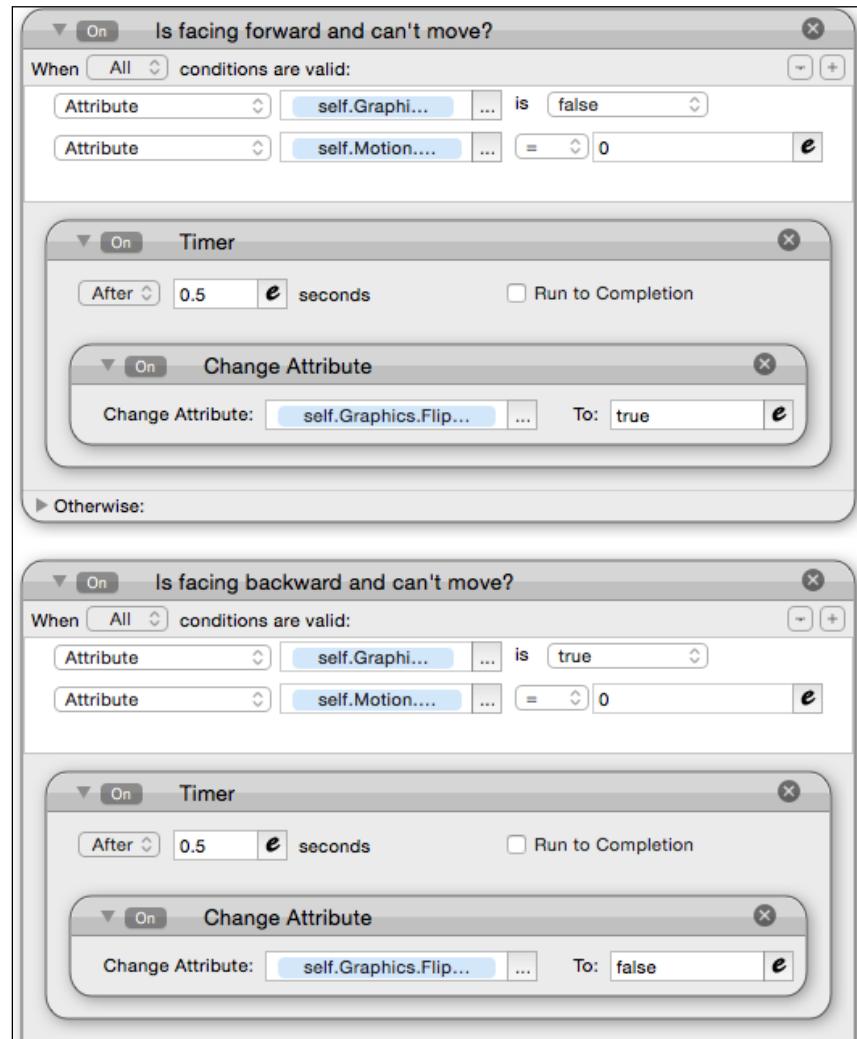
This is an easy way of moving our enemies around, but we are going to make things a little more complex. Why? Well, when the enemy hits a block, he will just stop. We don't want that. We want him to turn around and keep wreaking havoc:

1. Create a new rule. The rule will be entitled `Accelerate Forward`, which will be **Attribute** | `self.Graphics.Flip Horizontally` | **is** | **false**. Then drag the Accelerate behavior into this rule.
2. Copy and paste that rule, change it to **Attribute** | `self.Graphics.Flip Horizontally` | **is** | **true** and change the direction to 180.

Now we want to detect if he is stuck and can't move:

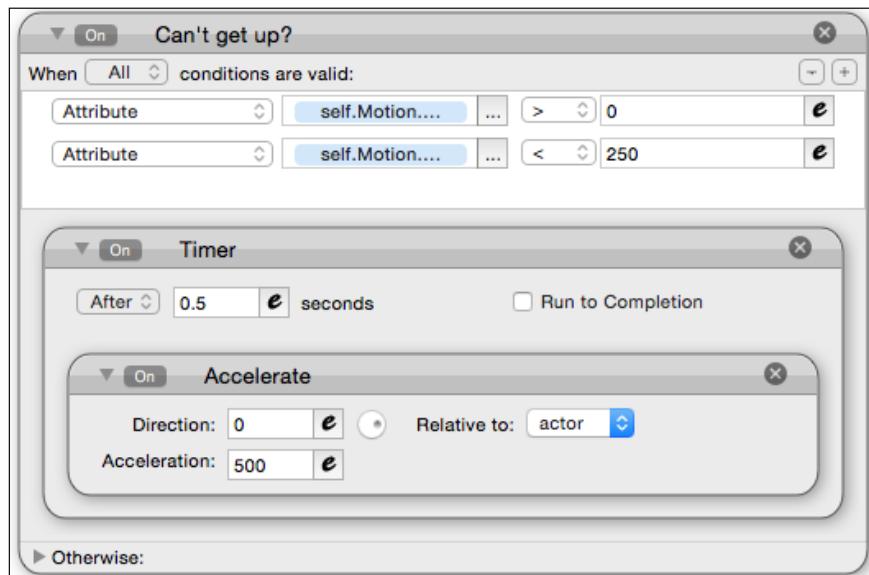
1. Create another rule, change the first condition to **Attribute** | `self.Graphics.Flip Horizontally` | **is** | **false**, click on the + button to add another condition, change that one to **Attribute** | `self.Motion.Linear Velocity.X` | **=** 0. This will now detect if he is facing forward, and not moving, that is he's stuck.

2. Next, drag in a Timer, set it to **After 0.5 seconds**, then drag in a Change Attribute behavior and change it to **Change Attribute: | self.Graphics.** **Flip Horizontally | to | true**. Now when you test it, when he hits an obstacle after 0.5 seconds, he will flip. Copy and paste that rule and change the conditions accordingly, so the initial rule will be true instead of false, and in the Change Attribute behavior, it will be false.



If you create a hill that is too steep, the enemy won't be able to make it up it using this method. No problem, we can fix this! Create another rule, name it **Can't get up?**. Change the first condition to **Attribute | self.Motion.Linear Velocity.X | > | 0**, and the second will be the same, but less than 250. This will detect if he is slowing down. Now another timer will be after 0.5 seconds. Inside the timer, drag an **Accelerate** behavior, keep the **Direction** to 0, and the **Acceleration** to 500. Test it out, he will now shoot up the hill!

Why the Timer there? While testing I found that, if he went down a hill and hit something, he immediately accelerated backwards without flipping. Everything needs some time to react and make sure it's not interfering with other conditions.



That's pretty simple, isn't it? It certainly removes a lot of time when programming our games. Think of all the lines of coding just to get our enemy moving and rotating! Not even including physics, collision detection, and our make-shift pathfinding.

If you want the character to stop moving before a crack in the ground, you can set up an invisible actor that will flip him. The behaviors would be the same, but instead of detecting if the linear velocity is 0, just detect the collision and flip accordingly.



Now let's get him attacking! This guy is going to fling sludge at poor Kevin, but don't worry! Kev can handle it.

## Attacking

I've created an animation for the sludge actor; so let's go ahead and import them into our project and create a new actor from the image. Inside the sludge actor, drag in an **Animate** behavior and drag in the other images so it looks like it's goopy as it flies through the air. Finally, drag in a **Timer** behavior, which will be **For 0 .2 seconds**, then drag in an **Accelerate** behavior and change the **Speed** to 2500.

Let's flip over to our enemy actor. We are going to drag in a **Timer** behavior, which will be **Every 2 seconds**. Inside the Timer, create two rules, one will be for forward facing, and the other backward facing. The condition on the forward rule will be **Attribute | self.Graphics.Flip Horizontally | is | false**, and the opposite for the backward rule. All you have to do is drag in a **Spawn Actor** behavior. Select the sludge actor.

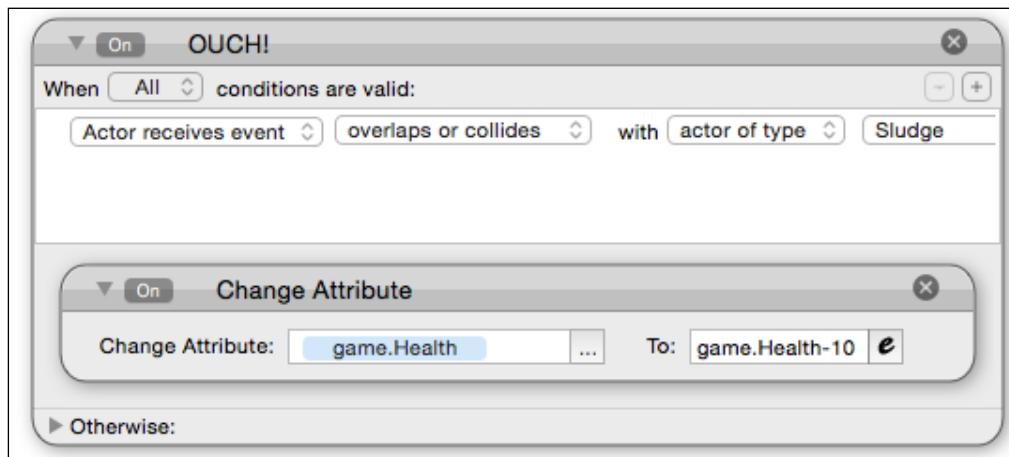
For the forward rule, change the **Direction** to 45, and the position to 50. For the backward rule, change the **Direction** to 135, and the Position to -50. This will spawn the sludge actor in an upper forward kind of motion.

**Give this a try**

Want to make a cool splash when the Sludge hits a platform or the floor? Try particles, or spawn an actor with an image of a puddle. See what you can do!



Now we need to chip health away from poor Kevin if he gets hit by one of these bad boys. Open up our Kevin actor, create a new rule which will be **Actor receives event | overlaps or collides | with | actor of type | Sludge**, then drag in a Change Attribute behavior and change that to **Change Attribute: | game.Health | to | game.Health -10**.



## Diving In - Enemies

---

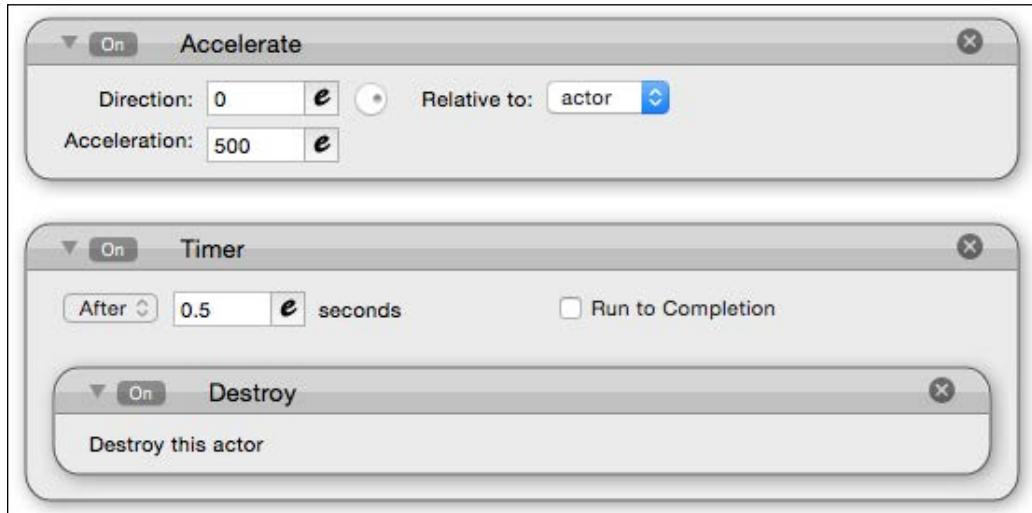
Now just go into the sludge actor and create a new rule, change it to **Actor receives event | overlaps or collides | with | actor of type | Kevin**, then drag in a Destroy behavior.

But I made this other guy, and he doesn't shoot. He lashes out with a swipe to attack. How would we do this? Create a new actor from the Walker images in this chapter's resource section. The animations are quite crude, but it's alright.

In the new Walker actor, copy and paste all the fun moving behaviors we had for the last bad guy. There are a few ways we could get him to attack. We could either have him spawn an actor that will detect collision with Kevin when he's attacking, or you could even have an invisible actor following in front of the walker to detect collision. The second option works well, but if you have more than one Walker on the scene, it won't work properly because the detector will be confused as to which Walker actor to follow.



We are going to go with the first option, so he is going to spawn an actor when Kevin touches him. It's a crude but effective way of doing it. Create a new object that will spawn from the walker actor. It will be movable, but will have `0 Alpha`, which will make it invisible because we don't want to see it. Then drag in an **Accelerate** behavior and change the speed to 500, then a **Timer** behavior with **After 0.5 seconds**, and inside the Timer drag in a **Destroy** behavior.

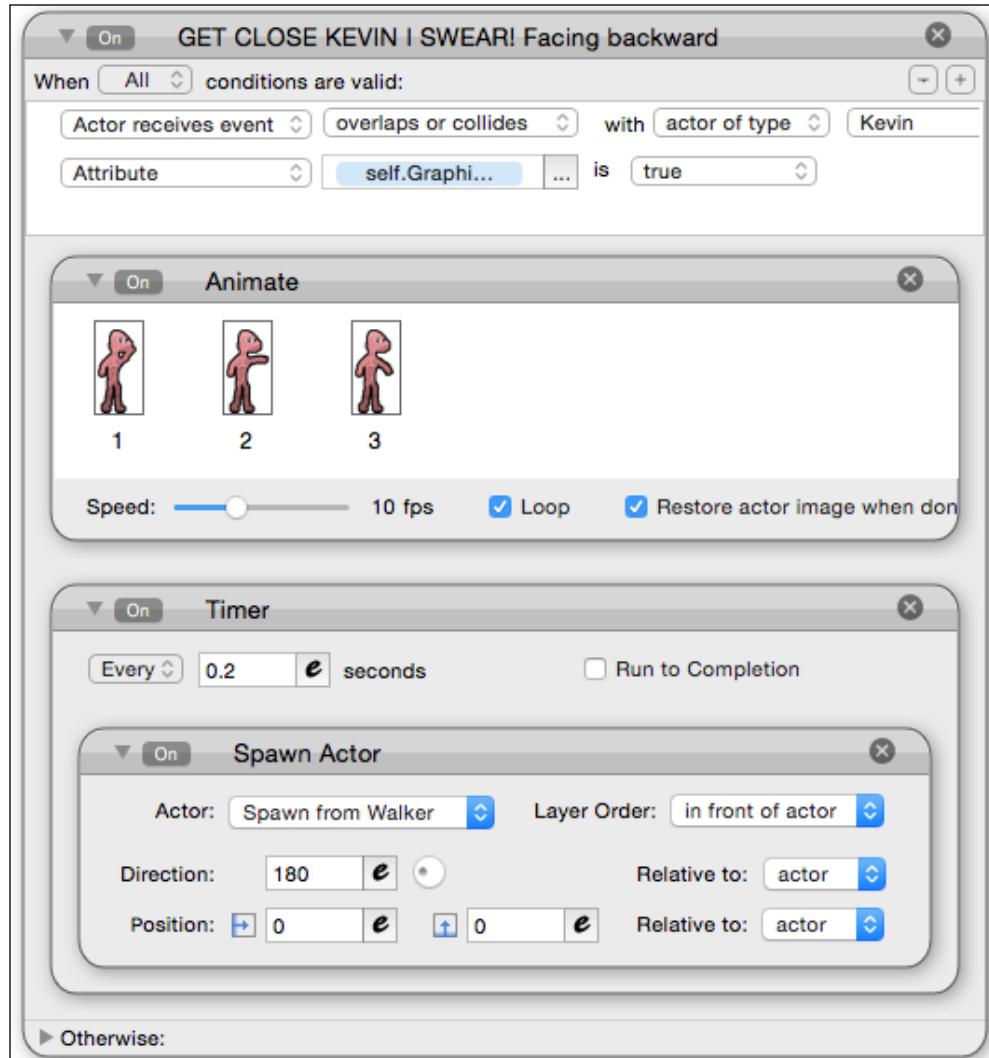


Now let's go back into our **Walker** actor, past all the movement behaviors. Let's create a new rule. This will be **Actor receives event | overlaps or collides | with | actor of type | Kevin**; add a second condition which will be **Attribute | self.Graphics.Flip Horizontally | is false**, then drag in an **Animate** behavior to show the attack, and then a **Timer**. Change the Timer to **Every 0.2 seconds**, then inside the Timer behavior, drag in a **Spawn Actor** behavior and select the actor you created to be spawned. Copy and paste this section of rules and change them accordingly for facing the opposite direction, **self.Graphics.Flip Horizontally | is true**, and change the direction of the **Spawn Actor** to 180.

## Diving In - Enemies

---

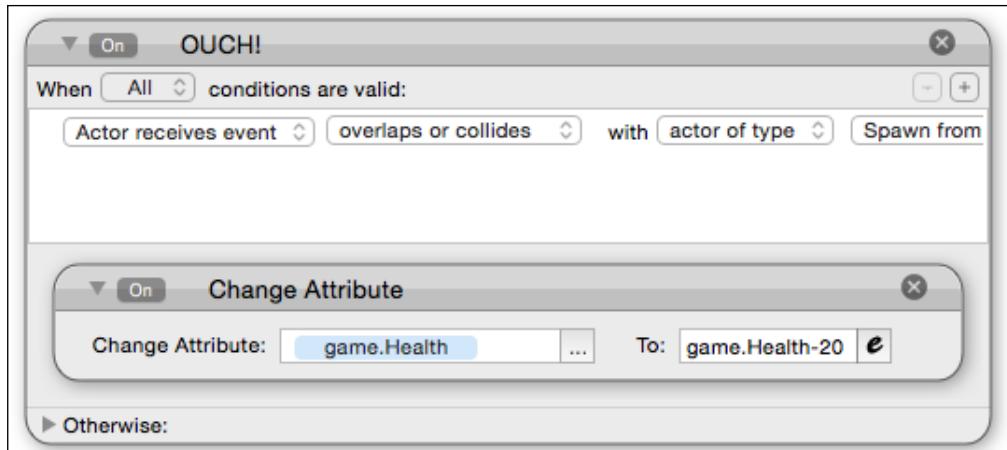
For testing purposes, change the **Alpha** of the spawned actor to 1 (so we can see if it is indeed spawning correctly), then test the level to make sure he is indeed tossing these objects.



Use the preceding screenshot to make sure you have input everything correctly. Like I've mentioned before, it's super easy to miss simple things, especially when you're programming for a long time.



If it does work, it should look like the immediately preceding screenshot. Now we have to make Kevin take damage. To do this, open up our Kevin actor, create a new rule (or copy the one from our Sludge collision rule) and change it to **Actor receives event | overlaps or collides | with | actor of type** | (insert spawn actor name here), and insert a Change Attribute behavior into this rule, change it to **Change Attribute: | game.Health | game.Health -20. Ouch!**

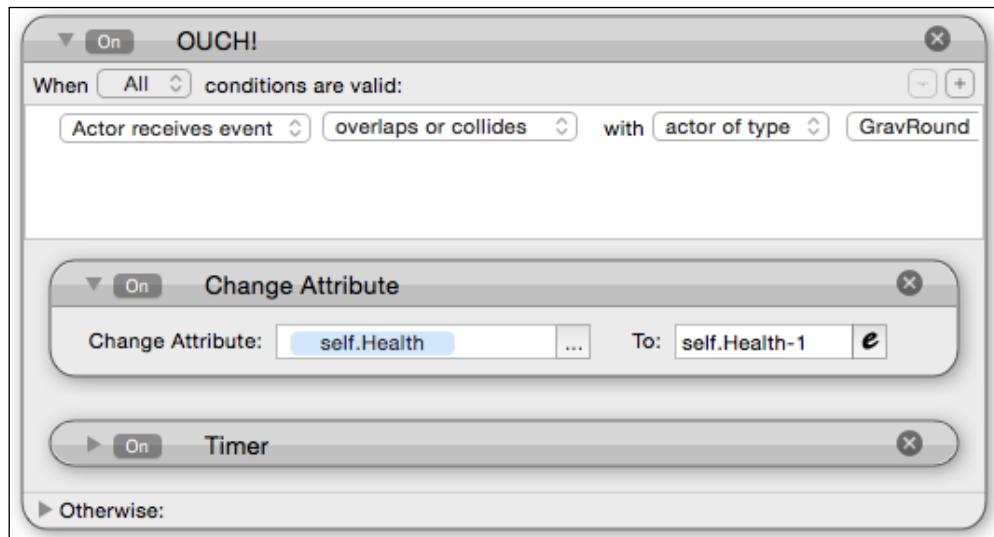


Change the **Alpha** on the spawn object back to zero and everything should be working perfectly! We have to give these baddies some health now!

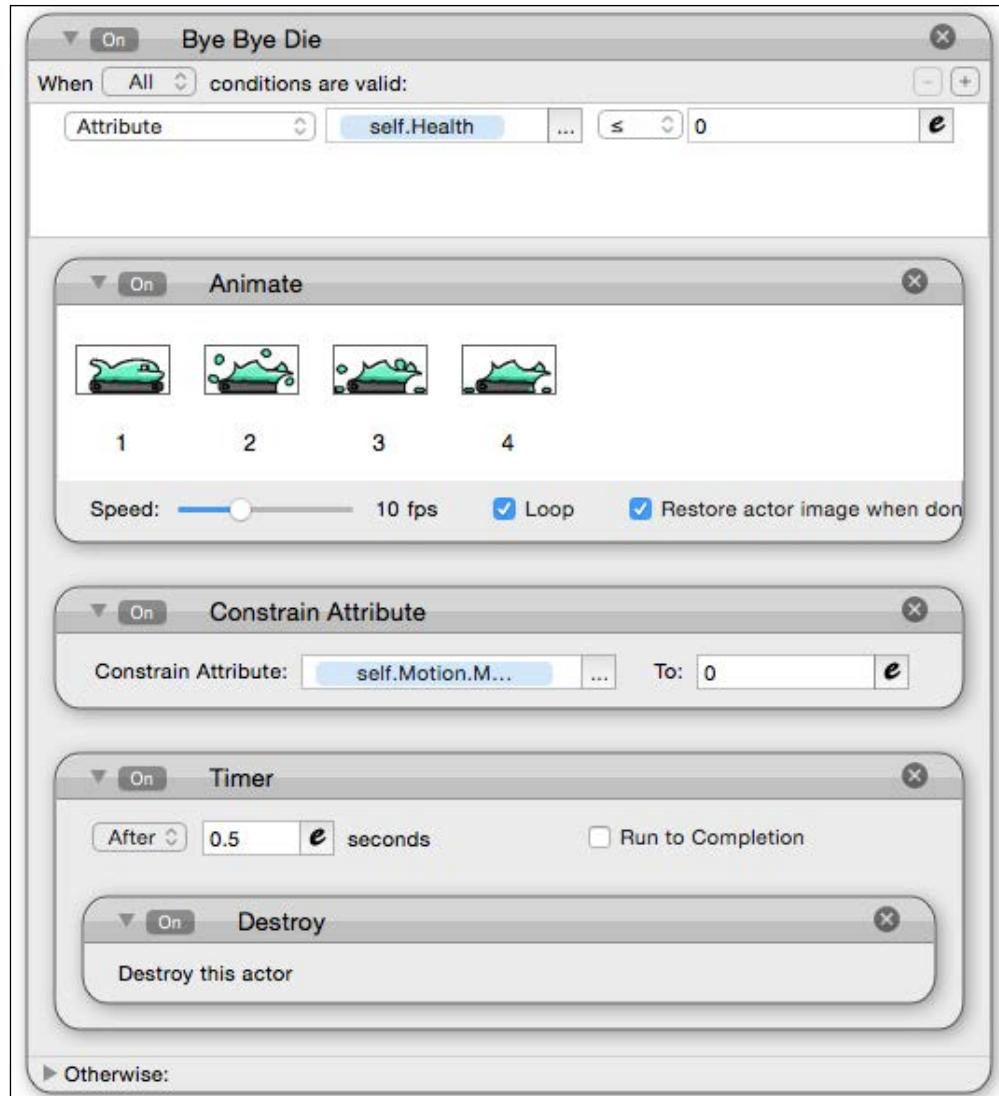
## Enemy health

As you've probably guessed, it's super easy to give the enemies health. You're right! We are going to follow the same procedure of giving Kevin health. Go into one of our enemy actors, and you guessed it, we are creating a new Integer attribute. Name it `Health`, and assign its value to 5 (or larger for tougher guys).

Now all we have to do is create a new rule, change it to **Actor receives event | overlaps or collides | with | actor of type | GravRound** (to avoid having to create this rule, numerous times you can create a new tag for all the projectiles and just detect collisions with actors of the tag), then drag in a Change Attribute behavior, set it to **Change Attribute: | self.Health | to | self.Health -1**.



Now we need to create another rule to detect if the health reaches zero. The new rule will be **Attribute | self.Health | < | 0** (when the health reaches zero, or goes past zero), drag in an **Animate** behavior and select the dying animations. We want to stop him from moving as well, so drag in a Constrain Attribute behavior, change it to `self.Motion.Max Speed | to | 0`, then add in a Timer behavior, change it to **After 0.5 seconds**. Finally, drag in a Destroy behavior.



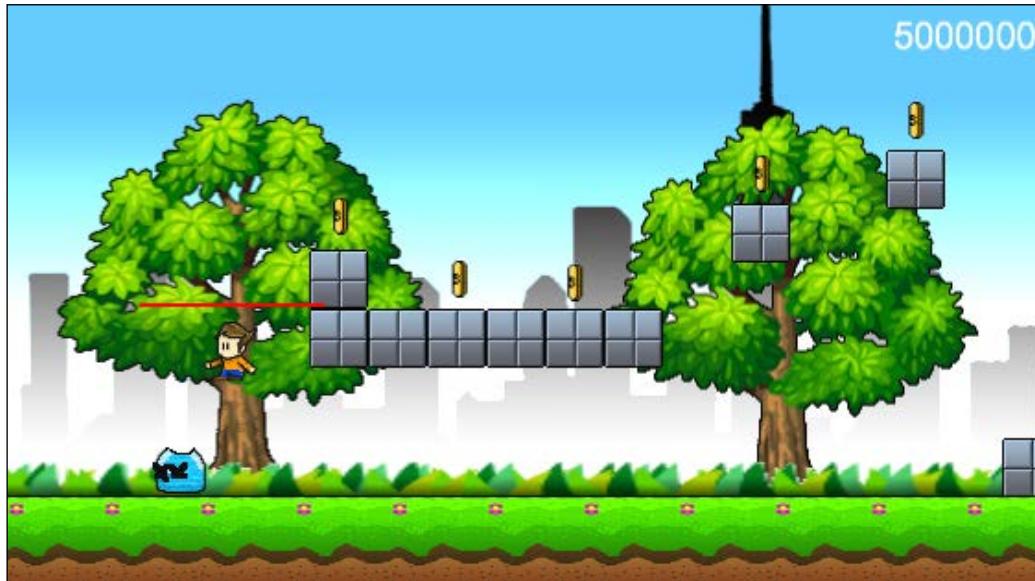
Repeat the same for all other enemy actors, unless you want a one-hit kill, like this guy:





### Want to do a Super Mario-style bouncing on the head?

Place an actor at your player's feet, and one on the enemy's head to detect a true collision. When a collision has occurred, kill the enemy and have a Timer on your player that will accelerate him upward for a short amount of time.



There are a lot of cool things you can do with enemies. In fact, if you were creating a shooting game, you could even have the enemies have a certain amount of ammo, and when they run out, they could either look for more or beat you with their guns; all this, just with a few little behaviors.

## Summary

Our game is getting more and more functional as we go on. Wow! We created some great enemies for our game. We even programmed a simple Artificial Intelligence system that helps them move around without pre-defining paths for them. Then we gave them health so that they can get hurt too! Let's be honest, Kevin is the one that is suppose to do the hurting, not just the enemies! Kevin has such a hard life.

We are almost done with the major mechanics of the game. In the next chapter, we are going to finish it up. We are going to look at doing some objectives, and then we will talk about what you can do with your game when you've finished reading this book.

On to the next chapter!

# 7

## Diving In – Objectives

We are coming to the end of this book, which means at the end of the book, you will have all the information and programming knowledge of GameSalad you need to be able to complete your game and release it to the public. Fun! Let's see what we are going to discuss in this chapter:

- Objectives
- Way finding
- Boss fights
- Multi-platform publishing
- Being awesome! (How to spread the word about your awesome game)

We are going to finish things up in our game and then we are going to send it out to the world. Let's dive right in!

### Objectives

Kevin will need to know what to do in each level, otherwise he would just be wandering aimlessly. That's not fun, unless you're playing **Skyrim** or **The Legend of Zelda**.

## Diving In - Objectives

---

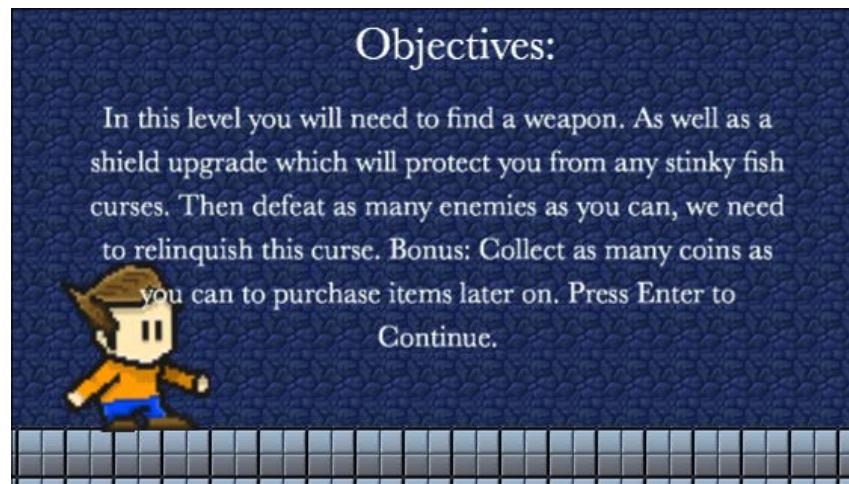
Let's start off easy. Let's add a new level just before our first level by clicking the + button at the bottom of the **Scenes** section. Then click and drag our new scene between the first level and the last part of the intro. This is going to be our objectives screen:



Lay out the background however you like. It doesn't have to be super fancy. Once you have laid it out, you are going to want to create a new actor. This actor will simply display text; that's all it will do. In order to do this, we need to turn **Alpha** down to zero in the actor's attributes, under the **Color** dropdown, and uncheck the **Movable** option. We won't add any behavior into the actor in the **Inspector** window because each of these actors we drag into the scene will read different text.

Now what we are going to do is add the text! I created two separate actors; one to display the title "Objectives" and a large actor with all the text with the player's objectives.

Is the text only displaying in one line? Then in the **Display Text** behavior, click on the **Wrap Inside Actor** option; this will wrap the text within the bounds of the actor.

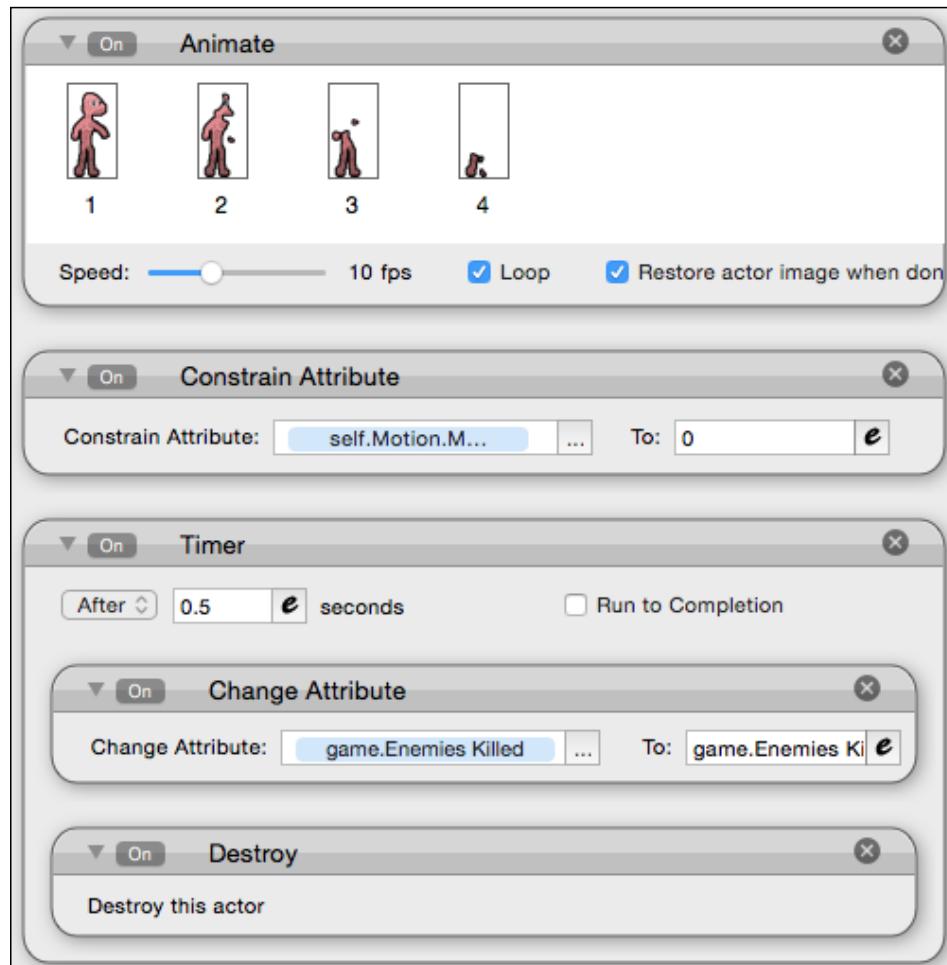


We are now going to set up some game-wide attributes for this objective. We already have attributes for the shield upgrade and the weapon collected, but let's create two Integer attributes; name the first `Enemies Killed` and the second `Coins Collected`. See where I'm going with this?

With these attributes, we are going to detect how many enemies were killed, how many coins were collected, and if the other objectives were completed, we are going to add to the score according to the success of the objectives in the level.

First, we need to detect how many coins have been collected and enemies killed. Now let's focus on the number of coins collected.

Go into our **Kevin** actor, and in the **Coin** rule, add a new **Change Attribute** behavior, and change it to **Change Attribute: | game.Coins Collected | to | game.Coins Collected +1**. Now each time a coin is collected, it will add one point to the attribute to be used at the end of the level. Now, for the enemies being killed, we will have to go into each of our bad guy actors, and under the rule where we detect zero health (I called my rules **Bye Bye Die**), do the same as with coins collected. Just add one instead to the **Enemies Killed** attribute:



Now we will create a new level just after our first level; in this level, the game will tally the score. Simply drag the text actors accordingly; one side for the labels (or objective description), and the second set displaying the corresponding information. For each coin collected and enemy killed, I'll add 10 points to the score, and for collecting the gun and shields, I'll add 100 points each.

Level Score:	
Gun Collected?	Yes!
Shield Collected?	Yes!
Enemies Killed:	1
Coins Collected:	17

Now let's add one last label in the center at the bottom that will tally up the score. This one will involve a slightly complex equation, but nothing difficult. First we are going to display the text.

We will need four different rules to detect each outcome:

- First, if the player collected the gun, but didn't collect the shields
- Second, if the player collected the shields but not the gun
- Third, if the player collected both
- Lastly, if the player collected neither

We will start with the rule where the player collected both the gun and shields. So create a new rule; it will have two conditions, which will be the following:

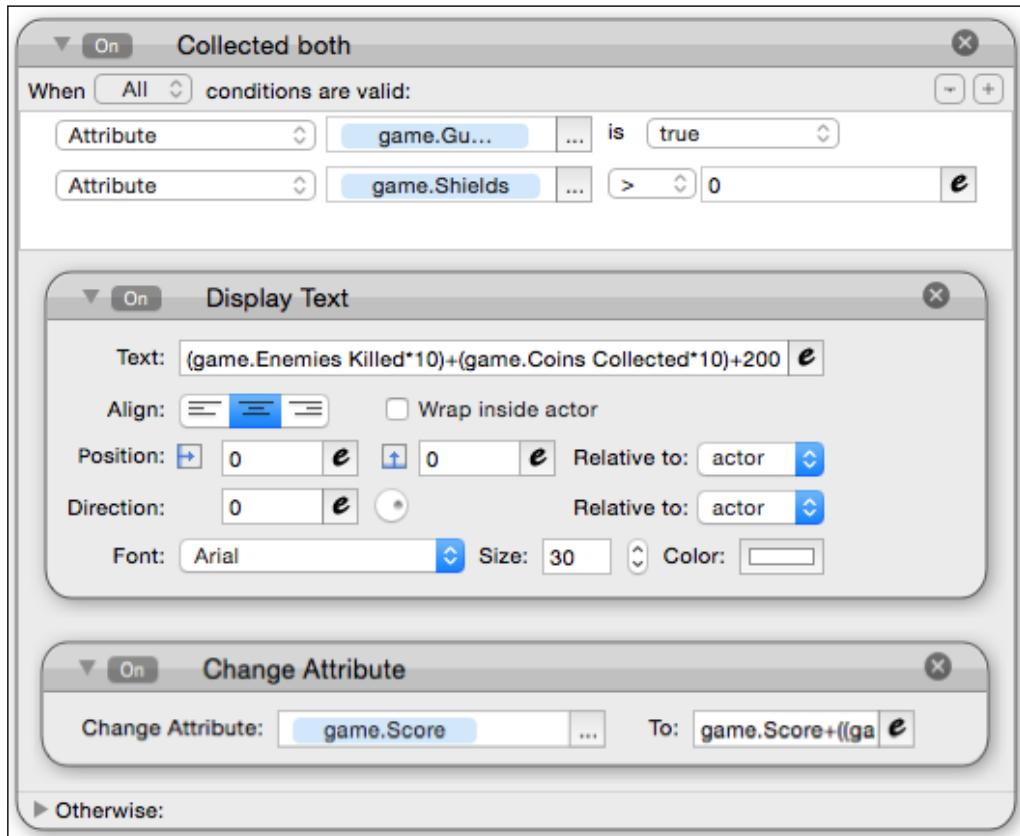
- First, **Attribute:** | game.Gun Collected | **is** | **true**
- The second condition will be **Attribute:** | game.Shields | **>** | 0

I did this because I didn't create a Boolean to detect whether they were collected or not.

Then drag in a **Display Text** attribute, click the **e** to open up the expression editor, and insert the following expression:

```
(game.Enemies Killed*10) + (game.Coins Collected*10) + 200
```

To do this, all you have to do is type in the brackets, then click the down arrow button and select each attribute accordingly, then type in the \*10, end brackets, and plus signs.



Then drag in a **Change Attribute** behavior, and change the score to the same expression as the Display Text behavior, which would read:

```
game.Score +(( game.Enemies Killed *10)+( game.Coins Collected *10)+200) .
```

## Diving In - Objectives

---

Now all you have to do is copy this rule and change the rule conditions according to each possible outcome of the level. So, for example, if the player somehow didn't collect the gun or shield upgrade, the rule would read:

```
Attribute | game.Gun Collected | is | false  
Attribute | game.Shields | = | 0
```

Then you will simply remove the +200 from the end of the expression. Alternatively, you can even add points according to how much health the player has left at the end of the level, and how many shields they have.



Now you can add in a **Timer** behavior to change the level, or have another label telling the player to press the *Enter* key to continue. When you call the **Change Scene** behavior, you will want to change the `Enemies Killed` and `Coins Collected` attributes to zero. This way the player isn't getting free points!

Test out the game to see if everything works, and if it does, at the end of the level, you will be brought to this screen and the game will add everything properly. Now you can copy and paste the objective screen and tally the score before and after each level, and adjust each objective however you like. You can even have an objective for the player to collect a set amount of coins, and if they do, reward them with 1,000,000 points. No, don't give them that much. They have to work for upgrades.

Now, on to way finding!

## Way finding

When you think of way finding, you may also think of signs inside a building that direct you where to go. I know that's what I think of, because I used to work in a signs shop. What we are going to do in this section is create a method of way finding with a GPS arrow that will direct you where to go. Let's say you wanted to direct the player to the position of the weapon they have to collect. Import the arrow image I have provided for you in this chapter's resource section, and then create a new actor from it. This actor *cannot* be movable, so make sure to uncheck the **Moveable** option in the actor's attributes. Obviously, we want to constrain the position of this arrow to the position of our player, so drag in two Constrain Attribute behaviors into the actor (for this part, we will be working on the actor in the **Inspector** window; we will edit the actor in the scene later on) and change the Constrain Attribute behaviors to `self.Position.X | to | game.KevX` and `self.Position.Y | to | game.KevY`.

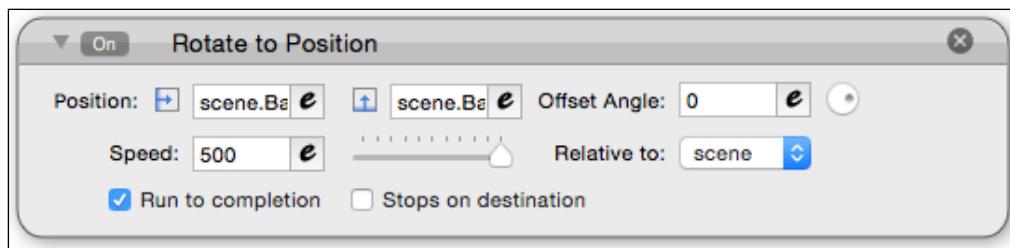
Finally, create a new Integer attribute inside the arrow and name it **Waypoint**. Then change its value to 1.

Now drag the actor into the scene and place it near our Kevin actor. Now we will edit the arrow actor in the scene. So double-click on it and click on the big lock icon.

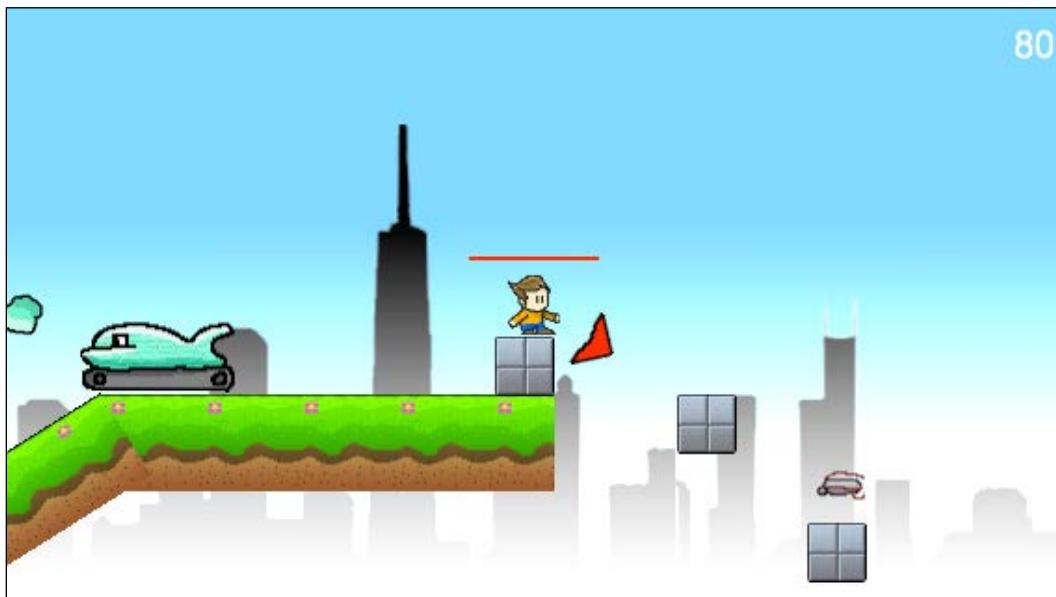
First we want the arrow to point to the gun, so let's drag in a **Rotate to Position** behavior. Apart from wanting the arrow actor to act differently at each level, the benefit of editing the actor within the scene allows you to also select and edit the attributes of the actor within the current scene.

Watch what I mean; for the **Position** option, click on the **e** side button to open up the expression editor, click on the down arrow to expand the expressions editor, click on **Current Scene**, and then click on **Layers | Background**. Then find the actor you want the arrow to point to and select its X position.

How awesome is that?! Repeat the same for the **Y** position. Finally, uncheck the **Stops on Destination** option in the **Rotate to Position** behavior. This will cause the actor to constantly look in that direction.



Test it to see if it works. If it does, awesome! Once the player collects the gun however, I want the waypoint to be destroyed because Kevin won't need it, and he will be able to navigate the rest of the level by himself. So let's insert a new rule and change it to **Attribute** | `game.Gun Collected` | **is** | **true**, then drag in a **Destroy** behavior. You can have multiple waypoints for the arrow to point to by having a single object the arrow will point to. So when the player has reached the waypoint, change the position of the object the arrow points at to where you want the player to go next. Simply create a new game-wide Integer attribute and have it set to the waypoint the player is currently on. Then in the object the arrow points at, have rules depending on which waypoint the player is on, and change the position accordingly. Here's a view of how the wayfinding arrow looks:



Now that we have a super easy way-finding system, Kevin has to end this! It's time for...

## Boss battles!

Kevin has gone through a super tough journey. He's fought many enemies, and now he is at **Magellan's Castle** to save Dom:



Here I've made a really cool scene where Magellan takes Dom away. Now, we have to program Magellan's movements and general unpleasants.

We will start by dragging our Magellan actor into our scene. I made him a little bit smaller to be more proportionate with Kevin, and gave him a new Integer attribute for his health at value 100. I also made a game-wide Boolean attribute called `Boss Battle`. This way, Magellan will only start moving once this Boolean is triggered. I think I want Magellan to be waiting for Kevin. Then once Kevin reaches the boss area, the battle will start.

I placed an invisible actor just in front of Magellan so that when Kevin collides with it, it triggers the `Boss Battle` Boolean attribute, and the battle begins.

Open up the Magellan actor. First and foremost, we need to change his `Maximum Speed`. I made it 500, which is a decent speed, considering that he'll be able to bounce around. Next, make sure his **Fixed Rotation** option is checked and his **Bounciness** is set to 1.

Now we will drag in a **Collide** behavior, and set it to **actors with the platform(s)**.

We will now create a new rule in which we'll build all our battle behaviors. The rule is **Attribute** | `game.Boss Battle` | **is** | **true**.

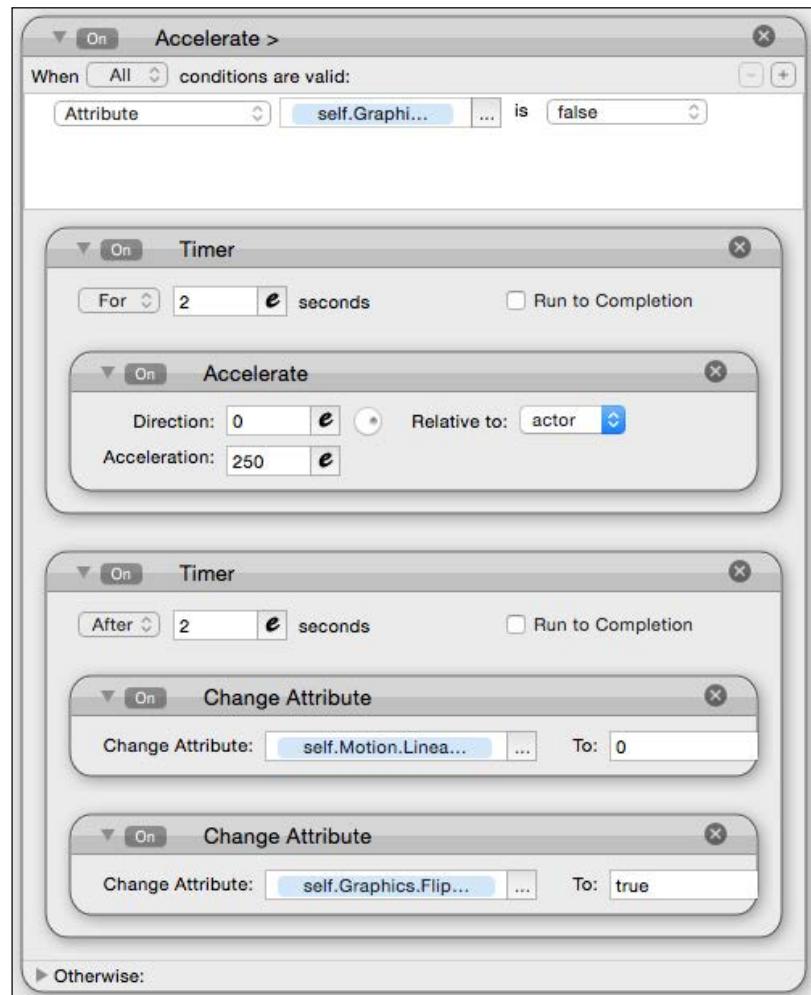
## Diving In - Objectives

---

First, we will add in a **Timer** behavior (one of many in this rule, believe me). This one will be **For 0.5 seconds**. Now drag in an **Accelerate** behavior, set the **Direction** to 90, **Relative to:** **Scene**, and the **Acceleration** to 5000.

Create a new rule and name it **Accelerate >**, and it will be **Attribute | self.Graphics.Flip Horizontally | is | false**.

Drag in two more Timers into this rule: the first one will be **For 2 seconds**. Inside this first Timer, drag in an Accelerate behavior, set the **Direction** to 0, and set **Acceleration** to **250**. The second Timer will be **After 2 seconds**. In the second Timer, drag in two Change Attribute behaviors; the first will be **Change Attribute: | self.Motion.Linear Velocity.X | to | 0**, and the second will be **Change Attribute: | self.Graphics.Flip Horizontally | to | true**.



Now just duplicate that rule we just created, and change it so the rule is **True**, **Accelerate** direction is 180, and change the last **Change Attribute** to **false** instead of **true**.

Finally, at the end of the rule, in the **Otherwise** dropdown, drag in two **Change Attribute** behaviors, both changing the **X** and **Y** values of `self.Motion.LinearVelocity` to 0, because we don't want Magellan to move before the boss battle begins.

Test it out to see if everything works properly. If it does, when Kevin reaches Magellan, the fight will begin with him shooting into the air and moving back and forth as he bounces off the ground.

We now want him to start attacking.

For Magellan's projectile, I recycled the star image by creating a new actor with it, and just changed the color. Inside the new **Star** actor, drag in a **Move** behavior, leave all the settings as the default values, and then drag in a **Particles** behavior if you want to get fancy.

Back to Magellan! Inside the **Accelerate** rule, drag in a **Timer** behavior, and change it to **Every 0.3 seconds**. Then drag in another **Timer**; inside it, drag in a **Spawn Actor** behavior and select the new **Star** actor we just created, and change the **Direction** to 315.

Do the same for the **Accelerate** rule, but change the **Direction** on the **Spawn Actor** to 225. Test it out to see if it works. If it does, he will be bouncing back and forth, shooting his stars at you as shown in the following screenshot:



Now all we have to do is program Magellan's health! To do this, all we have to do is create a new attribute named `Health`, with an Integer value of 250 or whatever figure you would like. Remember, this is a boss battle, so it has to be tough! Then create a new rule inside of the **Boss Battle** rule and make it **Actor receives event | overlaps or collides | with actor of type | GravRound** (or whatever projectile you want to use). Then drag in a Change Attribute behavior and change it to **Change Attribute: | self.Health | to | self.Health - 5**. Create a new rule to detect when his health reaches zero or less. Now destroy the actor with a **Destroy** behavior. You could do a cool animation of him dissolving, or him exploding in a bright blaze of glory.

Don't forget to change Kevin's loss of health as well when he gets hit by Magellan's stars. Do it the same way you did for Magellan!

If you want, when Magellan's health reaches a certain point, you can start a more aggressive attack as most boss battles do. You can have a new rule for this. When Magellan's health reaches 50 percent, start shooting more stars, or make him move faster and deal more damage.

That's it for the boss battle! Now that Kevin has defeated the evil Magellan and lifted the curse of the "Stinky Fish", he can release Dom from his holding cell. With what you've learned about cutscenes, why not create a cool celebration scene where the townsfolk are having a huge party to thank Kevin for his hearty work!

End credits.

Now that the game is finished, where do you go from here?

## Multi-platform publishing

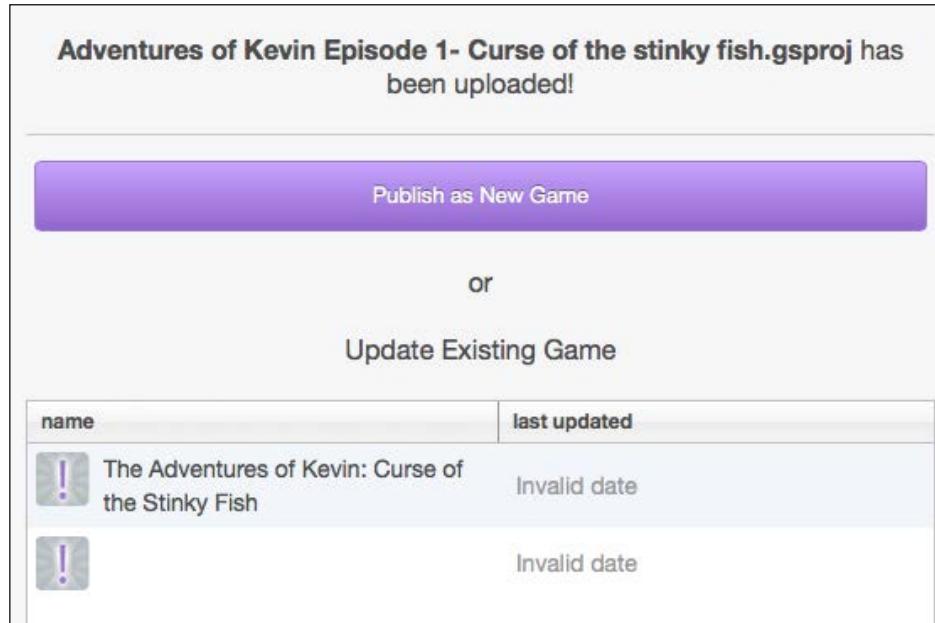
The great thing about GameSalad is you aren't limited to one platform, and you certainly don't have to change any settings between platforms.

Now that you have completed the development of your game, you want to publish it. Of course, with mobile platforms, you will need to be signed up as a developer. If you want to publish your title, simply click on the **Publish** button on the top bar of GameSalad, as shown in the following screenshot:



When you click on this button, GameSalad will prepare your project and upload it to their servers. The great thing about this is they store your project each time you publish it. So, if you have a mishap and you lose a bunch of data, you can go back and download the latest published project.

After the upload is completed, you will be brought to GameSalad's website, and once you log in, you will be prompted to create a new project, or update an existing game:



If this is your first time uploading your project, click on **Publish as New Game**. You will now be greeted with the page for your project. Here, you will be able to change the game icon, title, and set up all the builds:



Fill in the game title in the text field, and change the icon as well. Once that's done, you can click on the desired platform you want to publish your game to. I selected **Mac** for this example; and when you do, you will be asked to fill out some information, such as a nickname for the build, **Bundle Identifier** (usually `com.companynamespace.projectname`). For me, it is `com.wurdindustries.kevine1`, then the **Display Name** and version number.

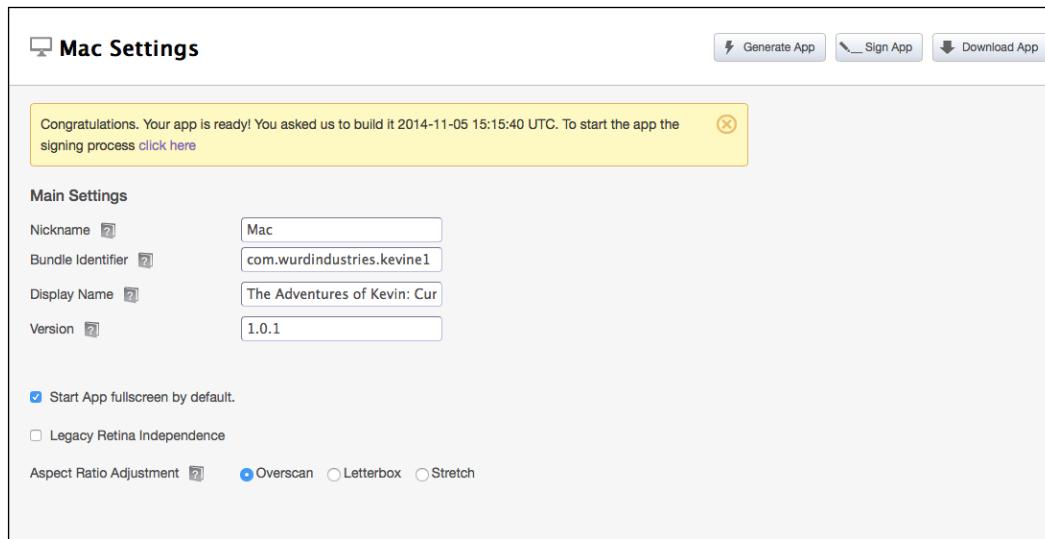
## Diving In - Objectives

---

Bundle identifiers are just that, a way to identify a bundle. You don't have to fill out forms or create different profiles for a bundle identifier; all you have to do is type it in. As mentioned previously, for me, it's `com.wurdindustries.kevine1`, but for you it could be `com.myawesomesomecompany.myawesomemeproject`. You name it whatever you like.

You will then be able to upload custom loading images and adjust app control settings such as starting the app in fullscreen as default, retina independence, and so forth. When you have all the information filled out, first click on the **Sign App** option (only needed for Mac and mobile devices) and then you can click on **Generate App**, which will generate the downloadable file. Once that's done, you can download the app by clicking on the **Download App** button.

From here, you can upload your game to your publishing site, such as **iTunes Connect** or to the **Android Store**.



Once this is done, you've done it! You have completed the development of your game! This isn't the end however, because at the time you release it, you won't make much money.

What's required next is telling the world about your awesome game!

## Spreading the word about your awesome game!

Releasing your game is certainly exciting, but it's not the end of your development. The most important thing you must do is market your game. Whether through Facebook, Twitter, Instagram, or even YouTube, spreading the word about your game is very important, or else it will get lost in the deep sea of other titles.

Send out posts, tweets, and videos about your great game; let everyone know! Advertising on these social websites is a fantastic way to spread the word as well. I advertised a trailer for my latest game **SpaceRoads**, and in one day, I got over 7000 views on YouTube. Facebook is another great advertising source, as there are millions of active users, and with these, you can target specific interests, ages, and genders who you believe will be interested in your game.

In one Facebook campaign that I ran for a week, I saw a huge surge in likes and purchases of my apps.

I've seen too many times a game get released where the developer doesn't advertise it, and after a matter of *hours*, the game gets lost in the list of new releases.

A few other things you must know when developing an app or a game:

- **Have a website:** Whether it's for your studio or just the game, it needs to be updated regularly. You also need to make the main page appealing and pop. If you don't, you will see a huge **bounce rate** (the time it takes for people to visit then leave your site).
- **Have a development blog:** This goes hand-in-hand with the website and social media outlets, but it's still important. Gamers love to read about the struggles and overcoming of obstacles that developers go through. (Phil Fish anyone?)
- **Create awesome trailers:** This comes later in the development, but an awesome looking trailer creates buzz and hype for gamers.

I've heard people recommending going to conventions and opening up booths. I've done it. I went to **Comic Con**, and I bought a stark 100 copies of SpaceRoads. By the end of the last day, I only had a few copies left, which I later gave away through a Facebook contest. Granted, this is a pricey way of advertising, as it cost me and Marc from **KFaction Productions** close to \$1000 to have the booth. Did the profit make up for it? Not really. My suggestion is to go to a place that specializes in gaming. **E3** and **PAX** are just a couple to mention (and that I have been close to opening a booth at).

Crowdsourcing is great too. Open up a kickstarter project, and if people see it being awesome, they will flock to you.

Contact the press. Don't go straight to IGN or GameSpot, but search for some indie game websites.

Press releases are also great. A website like PRWeb.com will distribute your press release to all the big guys like Bloomberg L.P., Google, and Bing, to mention a few, and the cost is fairly low.

Try Steam Greenlight. For a \$100 donation, you can upload as many games as you want. This can be huge! I currently have SpaceRoads as a Greenlight project, and I've been swamped with people wanting to play the game.

The most important thing you can remember is don't get discouraged. I've been developing for over 10 years and I've seen many of my projects go nowhere, and it's super easy to want to give up.

But when you get the big sales figures, all the failures are worth it.

## Summary

In this chapter, we discussed an excellent objective system, with the addition of a navigation and way finding system. We also discussed how to create a simple but effective boss fight, including movement for the boss, having him attack, and giving him health. Next we discussed multi-platform publishing; how you can deploy your awesome game to as many people as possible by utilizing GameSalad's multi-platform publishing system. Finally, we discussed how you can spread the word about your game. There are certainly a lot of options out there for you to get your game well known.

I hope you have enjoyed this book and learned a lot. Go out there and knock 'em dead my friends!

# The Challenges of Game Development

If you are a gamer, it's likely that you have dreamed of creating your own blockbuster game. No doubt all of us have, at one time or another, looked into developing our own game but are daunted by the fact that you have to learn complex programming languages. Granted, while most of these languages are easy to understand for some of us, most of us cannot understand them. We just do not want to take the time to scroll through endless tutorials and coding books to try and learn something. So when you are learning programming, do not forget about debugging!

However, the allure of seeing indie devs make it big may give you the drive and excitement to learn programming and game development.

To give you a better outlook on game development, we are going to discuss the following in this chapter:

- Examples of devs who have hit it big
- The challenges of development and the difficulty of learning coding languages
- How GameSalad makes it all easy

If you have the drive to start your own awesome game, you are going to want to read this. Not as discouragement, but to get a realistic look at the indie development scene.

## Examples of developers who made it big

Slumps... We all get in them. For good reasons too, especially in the game development world. It's so easy to get discouraged.

The excitement you feel when your game hits the store is unlike anything else. A few days go by and the sales are minimal, if nothing at all. Sure, we all get into a slump after a discouraging hit like that. Who wouldn't? Let me tell you, I was in a slump not too long ago. I had been working on my latest title *SpaceRoads* since my wife and I got married, 2 years ago. I put a lot of effort into it. I released trailers, gameplay videos, and screenshots to get hype for the game. I even paid to advertise my trailers. However, I didn't get the kind of response I was looking for. I got discouraged and halted development on the game.

I lost my job shortly after this happened, which left me even more discouraged. One day, feeling all doom and gloom, I turned on my **Apple TV**, loaded up some **Netflix**, and I started looking for a good laugh. Under the new releases section, I saw a documentary I had never seen before – *Indie Game: The Movie*.

By the time I finished watching that documentary, I was in tears. I felt everything the developers went through.

The movie follows the development of two of the biggest indie titles ever released: **FEZ** and **Super Meat Boy**.

## Phil Fish and FEZ

Let's start with Phil Fish, the Canadian developer of **FEZ**. No doubt you have heard his story.

Phil Fish had been developing **FEZ** for three years. Every one was hyping about it. The mechanics of the game were unseen and totally amazing! A 2D game in a manipulative 3D world. Because there was so much hype over the game, people got impatient to the point that people were getting very angry with Phil. They started to hate him. After four years of incognito (from the time he showed the prototype), Phil planned to show **FEZ** at **PAX (Penny Arcade Expo)** in Boston, a show that sees over 60,000 people in the weekend of the event. This was the first time anyone would be able to play **FEZ**. The pressure was *on*. Not only for Phil to finish a playable demo for the expo, but his former partner left the project and became involved in a huge legal battle.

Phil and his former partner had a matter of days to settle up; if they didn't, Phil wouldn't be able to show **FEZ** at **PAX**.

One signature, that's all that was needed. Everything was ready for PAX; the game was ready to go and his booth was printed and ready, but he needed that one signature to sign off on the project.

Imagine the frustration and stress one would feel in that situation. Who wouldn't freak out? You spent all that money, traveled from Montreal to Boston, and there's a risk of everything falling apart. One signature, one day left!

FEZ went on to PAX. Phil's partner agreed to release the project, and in the first day, FEZ sold 20,000 units. To compare, **Braid**, one of the biggest selling indie games before FEZ, sold 55,000 units in the first week.

All that frustration, all the errors and debugging and humongous headaches, but Phil Fish did it. That breath of fresh air when he saw the first sales figures!

To see what he went through, for me, was certainly an inspiration.



A screenshot from FEZ

## **Edmund and Tommy-Super Meat Boy**

The next game is what brought me to tears, because there were real tears that were shed in the movie.

**Super Meat Boy**, developed by Edmund McMillen and Tommy Refenes, was released on **Xbox Live Arcade** in 2010, with PC, OS X, and Linux releases to follow soon after.

Development started in January 2009, and the game was noticed immediately. Dubbed the most anticipated game of the year in 2010, it was certainly a big hit. Imagine again the excitement on release day as you go to turn on your Xbox to see your game on the front page... it's not there. A year of development, a highly anticipated game, killing themselves to get the game out, all for nothing. The stress the two went through. Tommy was almost a train wreck, and I understood.

The two devs were promised a front spot on Xbox Live Arcade; they rushed and pushed as hard as they could to reach the deadline from Microsoft, and they weren't there. Release day is the biggest sale date of any game; the majority of your sales will happen that day, and being promised a front spot on one of the biggest gaming outlets is a big deal! Eight hours into release day, the game was finally released. 10,000 units at 2:00 PM that day. Edmund said that in those ten hours, Super Meat Boy made more money than he did in the last six years put together.

Elation... Tears.



A screenshot from Super Meat Boy

The sacrifices these two guys made to make the game were all of a sudden worth it. Here's what got me. As Edmund and his girlfriend were watching the reviews, his girlfriend started crying, and felt that the sacrifices finally seemed worth it. It gets even better. Tommy, whose family made huge sacrifices for him, including remortgaging their house, paid off his parents' debt.

Edmund and his girlfriend bought a house, and a hairless cat.

Soon after that, Tommy and Edmund started working on their next game. Who wouldn't, after such a success?

Granted, these stories don't happen to everyone, but to know that the biggest indie developers to make it huge went through tribulation, and spent their blood sweat and tears, never giving up and going from having absolutely nothing to being able to pay off their parents' debts, for me is encouragement enough.

Watch the documentary. It's amazing. (There is a lot of language in it, so beware!)

## **The challenges of game development**

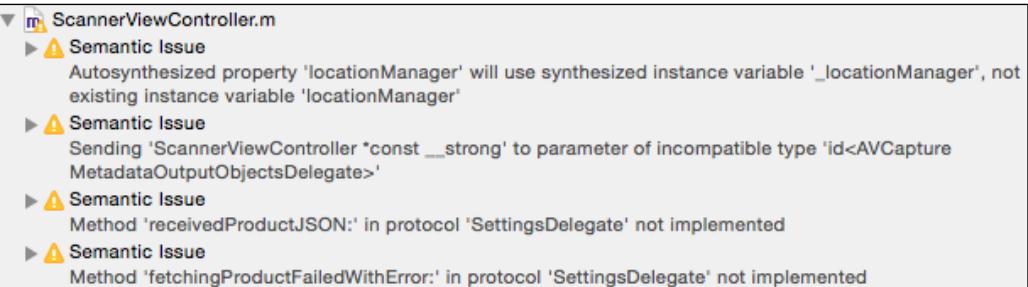
If you've watched the *Indie Game: The Movie* documentary, you will have seen just some of the challenges of game development. Game development is mentally, physically, and emotionally taxing. Especially considering people invest financially and emotionally into their projects. They sacrifice hours of family time, or any spare time at all to develop their game. Some have no job and run off savings, or in the case of some, live off their parents' or spouse's income.

The stress of development too is very difficult. For the most part, the workload that around 200 people do is put on the shoulders of one or two people. So you have to consider that not only are you programming a game, but you are creating all the images, sound effects, and music. That's a lot of work! There have also been many times when developing a game, you get so intensely into it that nothing else matters. Then you get a game killing bug. Then there's the stress of the interruption, and the stress of finding the error, which can take many many hours to locate.

Now let's talk about the difficulty of learning a new programming language.

Some languages are a little less complex than others, but for the most part, learning a new language is exactly that. You are learning a new language. Did you ever take a secondary language in school? I took French and Spanish and I picked up Spanish much faster than I did French. Why? Some languages are easier than others. It still takes a lot of time and effort to study.

How long, and how much did you study your language courses? The same amount of effort is required for learning a coding language. It's tough work. Who has the time either? Courses can be costly too, costing around \$10,000 per course, and tutorials are helpful but not always easy to follow as most of them assume you know the program and language. If something goes wrong, you are kind of out of luck because there's no explanation. Sure you can search forums and post comments, but who's to say you'll get a response?



What the heck does that mean?! Ah, syntax and semantic errors. Now this is where this book comes in handy...

## How GameSalad makes it all easy

GameSalad simplifies the development of your game. No need to learn a crazy confusing language! All the *programming* you do here is done by drag-and-drop behavior-based modules into the actors or objects in the game. Everything is drag-and-drop. No need to program tiles to build your levels!

For example, here is a very simple collision detection system using the **iOS Sprite Kit** development:

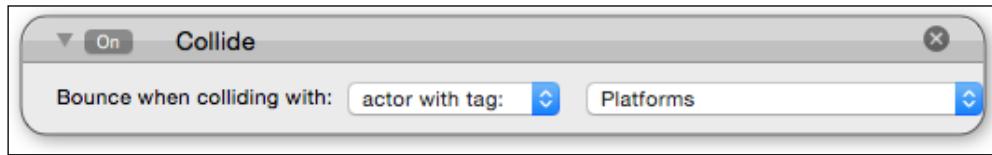
```
- (void)checkForAndResolveCollisionsForPlayer:(Player *)player
forLayer:(TMXLayer *)layer
{
    NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};
    player.onGround = NO; //Here
    for (NSUInteger i = 0; i < 8; i++) {
        NSInteger tileIndex = indices[i];

        CGRect playerRect = [player collisionBoundingBox];
        CGPoint playerCoord = [layer
            coordForPoint:player.desiredPosition];
```

```
NSInteger tileColumn = tileIndex % 3;
NSInteger tileRow = tileIndex / 3;
CGPoint tileCoord = CGPointMake(playerCoord.x +
    (tileColumn - 1), playerCoord.y + (tileRow - 1));

NSInteger gid = [self tileGIDAtTileCoord:tileCoord
    forLayer:layer];
if (gid != 0) {
    CGRect tileRect = [self
        tileRectFromTileCoords:tileCoord];.....
```

It goes on for another three pages. Now let's see how to detect collisions in GameSalad:



Well that was easy now wasn't it? That's the gist of developing in GameSalad!

Another issue with game development is **platform targeting**. For the most part, each platform has a different scripting language. So let's say for iOS you need to learn Swift or C#, with Android it's C or C++, and with **Windows Mobile** it's **Visual Basic** (it doesn't drink pumpkin spice lattes) or C++ as well. While a lot of the languages are similar, it involves using different SDKs to develop on these platforms. GameSalad removes the need for that. In fact, GameSalad allows multi-platform publishing with the click of a button. No need to reprogram everything.

I know what you're thinking. GameSalad can't do what other SDKs can do.

GameSalad is just as powerful as any other SDK. It can do complex equations, physics, image effects, and all the cool stuff. It's inexpensive too! Let's take **Unity3D** for example; to buy it with all the publishing support that GameSalad has, you are looking at around \$4,500. For the **Unreal Engine**, you're looking at over \$200 a year, *plus 5 percent of sales*. GameSalad? \$299 a year! Flat. All this for an easy-to-use, powerful engine with no programming.

GameSalad it is! Now I know I'm comparing apples to oranges here, as GameSalad can't do super awesome looking 3D graphics, but it can do awesome games. If you harness the power of GameSalad, you can do some awesome looking stuff. Super Meat Boy on GameSalad? No problem! FEZ? Difficult, but you could probably pull it off, or something very similar to it. Final Fantasy? Again, super easy. GameSalad is very powerful and can handle almost any 2D project you throw at it. So go ahead; grab game development by the horns and give it a go. Whether you continue to use GameSalad or move on to other engines, the world of game development is a fun one, albeit a cut throat one. The potential in the game development world is huge. Just search for indie game earnings in Google and it's enough to make you shiver. Are you thinking, "I can make that much money just by typing?" Yes, you can if you really put your mind to it. Don't let anything deter you, and don't get discouraged when things don't pan out the way you want. It happens.

Another thing that will happen is summed up in the following picture.



You are going to create many many projects that you probably won't finish. Sometimes, a project is awesome when you start it, but after a while it stagnates and you just have to move on to a new one. I don't know how many projects I've started and haven't completed, but I know it's too many to count.

It's a fun world out there! Go develop all you can imagine and have fun. Thank you for reading this book.

# Index

## A

Accelerate behavior 60  
Actors tab, GameSalad 14, 15  
ammo  
    about 86-97  
    counting 90  
Android Store 126  
Animate behavior 40  
app development  
    prerequisites 127

## B

Bing 128  
Bloomberg L.P. 128  
bounce rate 127  
Braid 131  
Bundle Identifier 125

## C

Change Attribute behavior 53  
Change Scene behavior 6, 44  
Change Table Value behavior 83  
Chartboost 6  
Collide behavior 62, 100  
Control Camera behavior 51  
controls  
    about 59-65  
    mobile controls 65

## D

design process  
    about 20  
    baseline template 20  
    storyboards 21, 22  
developers  
    examples 130  
    FEZ 130  
    Phil Fish 130  
Display Text behavior 43, 81

## E

emotion  
    adding, to scene 57, 58  
enemies  
    attacking 104-109  
    creating 99, 100  
    health 110, 112  
    motion 101-103  
    path finding 101-103  
eye drop tool 25

## F

FEZ 130, 131

## G

game development  
    challenges 133-136  
    prerequisites 127  
game  
    introduction, creating 49-56  
    saving 81-85

## **GameSalad**

account, need for 9  
downloading 7-9  
forums, URL 84  
installing 7-9  
marketing 127, 128  
objectives 114  
project, creating 29-35  
publishing, on multi-platform 124-126  
starting with 29  
tabs 13  
URL 54  
version 0.11, features 5

## **GameSalad account**

Mac installation 9  
need for 9  
Windows installation 10

## **GameSalad, tabs**

Actors 14, 15  
Scenes 13  
Tables 16, 17

## **GameSalad version 0.11**

features 6  
interface 10-13

## **H**

### **health system**

creating 78-80

## **I**

### **installation, GameSalad**

Inter Dimensional Gravity Disruptor 70  
interface, GameSalad version 0.11 10-13

### **inventory system**

creating 86-89

### **iOS Sprite Kit development**

iTunes Connect 126

## **L**

### **levels, main menu**

selecting 47, 48

### **logo animation**

37-44

## **M**

### **Mac installation, GameSalad account**

#### **main menu**

creating 44-47  
levels, selecting 47, 48  
**melee weapons** 74, 75  
**mobile controls** 65-70  
**Move behavior** 40  
**Move To behavior** 40, 42

## **O**

### **objectives, GameSalad**

## **P**

### **paint bucket tool**

### **parallax backgrounds**

### **particles**

about 56  
working with 56

### **Particles behavior**

### **Phil Fish**

### **platform targeting**

### **player movement**

about 59-65  
attacking 70-74

### **power ups**

armor power up 86  
performing 86

### **project, GameSalad**

creating 29-35

## **R**

### **Reset Scene behavior**

### **RevMob Ad Networks**

### **Rotate to Position behavior**

## **S**

### **Save Attribute behavior**

### **Save Table behavior**

### **scene**

emotion, adding 57, 58

**Scenes tab** 13, 14

**scoring system**

    creating 81-85

**SpaceRoads game** 127

**sprites**

    about 23

    drawing, from screen to paper 23-29

**storyboards** 21, 22

**Super Mario-style** 112

**Super Meat Boy game** 131-133

## **U**

**Unity3D** 135

**Unreal Engine** 135

## **W**

**way finding** 119, 120

**Windows installation, GameSalad**

    account 10

**Wrap Inside Actor option** 114

## **T**

**Tables tab, GameSalad** 16-17

**tiling** 28





## Thank you for buying GameSalad Essentials

### About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

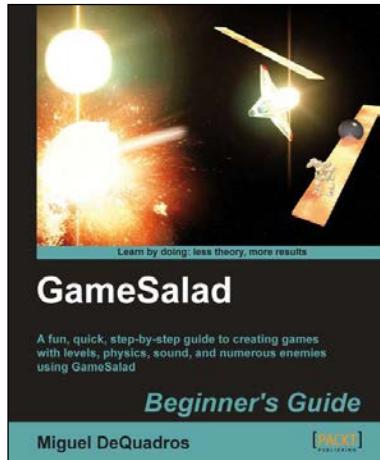
### About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

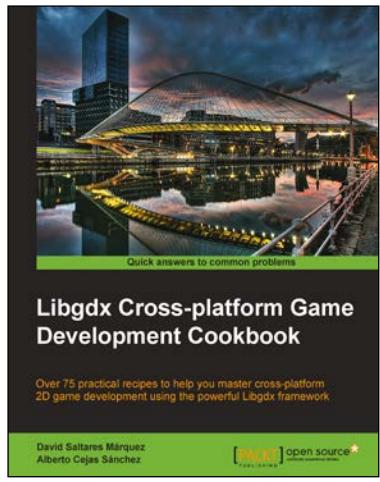


## GameSalad Beginner's Guide

ISBN: 978-1-84969-220-5      Paperback: 308 pages

A fun, quick, step-by-step guide to creating games with levels, physics, sound, and numerous enemies using GameSalad

1. Learn to build three games; Ball Drop, Space Defender, and Metal Mech with GameSalad.
2. Complete these games with sound effects, use of physics, gravity, and collisions.
3. Learn tips and tricks to make a game popular straight from the author's own experience.
4. Follow a step-by-step, tutorial-focused process that matches the development process of the game with plenty of screenshots.



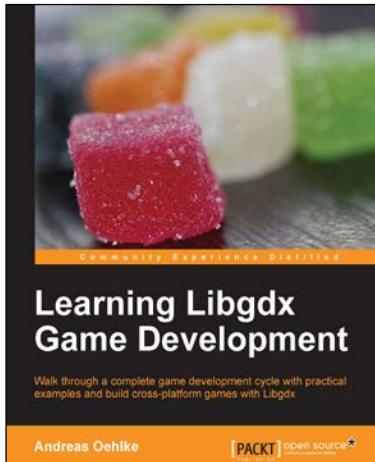
## Libgdx Cross-platform Game Development Cookbook

ISBN: 978-1-78328-729-1      Paperback: 516 pages

Over 75 practical recipes to help you master cross-platform 2D game development using the powerful Libgdx framework

1. Gain an in-depth understanding of every Libgdx subsystem, including 2D graphics, input, audio, file extensions, and third-party libraries.
2. Write once and deploy to Windows, Linux, Mac, Android, iOS, and browsers.
3. Full of uniquely structured recipes that help you get the most out of Libgdx.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



## Learning Libgdx Game Development

ISBN: 978-1-78216-604-7

Paperback: 388 pages

Walk through a complete game development cycle with practical examples and build cross-platform games with Libgdx

1. Create a libGDX multi-platform game from start to finish.
2. Learn about the key features of libGDX that will ease and speed up your development cycles.
3. Write your game code once and run it on a multitude of platforms using libGDX.
4. An easy-to-follow guide that will help you develop games in libGDX successfully.



## Building Web and Mobile ArcGIS Server Applications with JavaScript

ISBN: 978-1-84969-796-5

Paperback: 274 pages

Master the ArcGIS API for JavaScript, and build exciting, custom web and mobile GIS applications with the ArcGIS Server

1. Develop ArcGIS Server applications with JavaScript, both for traditional web browsers as well as the mobile platform.
2. Acquire in-demand GIS skills sought by many employers.
3. Step-by-step instructions, examples, and hands-on practice designed to help you learn the key features and design considerations for building custom ArcGIS Server applications.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles