Beginning Mobile App Development with Corona

Brian G. Burton, Ed.D.

Beginning Mobile App Development with Corona

Brian G. Burton, Ed.D.

Beginning Mobile App Development with Corona
By Brian G. Burton, Ed.D.

Copyright © 2011 Brian G. Burton, Ed.D. All rights reserved.
Printed in the Abilene, Texas, United States of America

Published by Burtons Media Group.

Electronic editions are available. See http://www.BurtonsMediaGroup.com/books for more information.

Corona® SDK is a registered trademark of Ansca® Inc. Ansca, the Ansca Logo, anscamobile.com are trademarks or registered trademarks of Ansca Inc.

Cover images were generated using Corona Simulator and represent views of apps made in this book on the Droid®, Galaxy Tab®, iPad®, and iPhone® (from left to right).
Trademarked names and images may appear in this book. Rather than use a trademark symbol with every occurrence, we have used the name only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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

All SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

ISBN (Print): 978-1-937336-03-5 | 1-937336-03-4
ISBN (eTextbook): 978-1-937336-02-8| 1-937336-02-6

Version 1.0

Quick Table of Contents

About the Author

Foreword by Carlos Icaza

Preface

1) Hello World: Setup and Get Going!
2) Buttons and Text
3) Animation, Alpha & Orientation
4) Fill in the Blanks
5) All Things Graphic
6) User Interface
7) Application Views
8) Phun with Physics
9) Creating a Game with Corona
10) Star Explorer Continued
11) Media Makes the World Go Round
12) File Storage & SQLite
13) Waiting on Tables
14) It's Who you Know: Networking
15) Working with Widgets & Popups
16) Rotten Apple – a Tower Defense Game
17) Additional Resources

Appendix A: The Lua Language
Appendix B: Advanced Lua

Extended Table of Contents

Quick Table of Contents IV

About the Author X

Foreword XI

Preface XIII
- **Welcome** XIII
- **Who This Book Is For** XIII
- **How This Book Is Organized** XIII
- **Conventions Used In This Book** XIV
- **Using Code Examples** XIV
- **Why didn't I use _____ for _____** XIV
- **Appendices** XIV
- **How to Contact Us** XIV
- **Why I Chose to Self-Publish** XV

Chapter 1 1

Hello World: Setup and Get Going! 1
- **What this book is not** 1
- **Getting Started** 1
  - Corona: Some background 1
  - Software: Corona 2
  - Software: Android 2
  - Examples and Graphics 2
  - Software: iOS 2
- **Development Hardware: Corona** 2
- **Development Hardware: Test Devices** 3
  - Android 3
  - *iOS* 3
- **Publishing Considerations** 4
  - Android (Google) 4
  - iOS (Apple) 4
- **Programming in Corona: Lua** 4
- **Programming in Corona: Editors** 5
- **Configuring Corona** 6
  - Macintosh 6
  - Windows 7
- **Programming in Corona: Hello World (V1.0)** 8
- **Project 1.0: Hello World** 8
  - Project Setup 8
- **Project 1.1: Hello World (v2.0)** 12
  - Objects 13
  - Summary 14
  - Assignments 14

Chapter 2 15

Buttons and Text ... 15
Know your Boundaries ... 15
Project 2: Button Fun ... 16
Functions ... 17
Project 2.1: Button Fun V2 ... 19
Getting Fancy! ... 20
How Corona reads your main.lua file ... 21
Summary ... 21
Assignments ... 22
Chapter 3 ... 23
Animation, Alpha & Orientation ... 23
Animation ... 23
Project 3: Basic Animation ... 23
Now You See It, Now You Don't ... 25
Project 3.1: Alpha Fun ... 26
Orientation change ... 28
Project 3.2: A New Orientation ... 29
Summary ... 33
Assignments ... 33
Chapter 4 ... 34
Fill in the Blanks ... 34
TextField ... 34
Project 4: Simple Calculator ... 34
Device Builds ... 41
Apple iOS ... 41
iOS Simulator Build ... 42
Apple iOS Device Build ... 42
Android OS Device Build ... 44
Assignments ... 46
Chapter 5 All Things Graphic ... 48
Vector Graphics ... 48
Project 5: Vector Shapes ... 49
Bitmap Graphics ... 51
Resolution ... 52
Scaling ... 53
Masking ... 53
Sprite Sheets ... 54
Project 5.1: Uniform Sprites ... 56
Project 5.2: Non-uniform Sprites ... 57
Summary ... 59
Assignments ... 59
Chapter 6 ... 61
User Interface ... 61
Resources ... 61
build.settings ... 61
config.lua ... 62
Dynamic Content Alignment ... 63

Dynamic Image Resolution 63
UI.Lua 64
Adding Sound 65
Sound File Types 65
Timing Is Everything 65
Streams and Sounds 65
Project 6: Beat-box 66
config.lua file 67
build.settings file 68
Summary 71
Assignments 72

Chapter 7: 73

Application Views 73
Hiding the Status Bar 73
Groups 73
Project 7: Group Movement 74
Modules and Packages 74
Project 7.1: External Library 75
External Libraries 76
CrawlSpace 76
Director 76
Project 7.2: Creating a Splash Screen 76
Summary 79
Assignments 79

Chapter 8: 81

Phun with Physics 81
Turn on Physics 81
Scaling 81
Bodies 82
Body Types 82
Density, Friction, and Bounce 82
Body Shapes 83
Body Properties 83
Body Methods 84
Project 8: Using Force 85
Gravity 89
Ground and Boundaries 89
Project 8.1: Playing with Gravity 89
Collision Detection 92
Sensors 93
Joints 93
Pivot Joint 93
Distance Joint 94
Piston Joint 94
Friction Joint 94
Weld Joint 95
Wheel Joint 95
Pulley Joint 95
Touch Joint 96

Common Methods and Properties for Joints 96
Project 8.2: Wrecking Ball 96
Trouble Shooting Physics 98
Summary 98
Assignments 99

Chapter 9: 101

Creating a Game with Corona 101
Game Design 101
Dragging Objects 103
Collision Detection 106
Take Your Best Shot 108
Reducing Overhead 108
Game Loop 109
Summary 110
Assignments 110

Chapter 10: 112

Star Explorer Continued 112
Configuring the App for Multiple Devices 112
Splash Screen 113
Improving Performance 118
Varying Difficulty 124
Increasing Game Speed 124
A Little Variety 125
Summary 125
Assignments 126

Chapter 11: 127

Media Makes the World Go Round 127

(or Can You Hear Me Now?) 127
Working with Sound 127
Basic Audio Controls 127
Duration Audio Controls 128
Volume Controls 129
Audio Channels 129
Sound File Types (Revisited) 130
Where did I put that file? 130
Multimedia API 130
Recording Audio 131
Project 11: Simple Audio Recorder 131
Video Playback 138
Camera 138
Project 11.1 X-Ray Camera 138
Summary 142
Assignments 142

Chapter 12: 145

File Storage & SQLite 145
File IO Considerations 145

Reading Data 146
Implicit vs. Explicit File Manipulation 146
Implicit Read 146
Explicit Read 146
Writing Data 147
Implicit 147
Explicit 147
JSON 147
SQLite 147
LuaSQLite Commands 148
Project 12: Reading a SQLite Database 148
Project 12.1 Writing to a SQLite Database 151
Summary 161
Assignments 161
Chapter 13: 163
Waiting on Tables 163
Table vs. Table: Clearing up the Confusion 163
Tools for Tables 163
Project 13: Creating a Simple Table View 164
Project 13.1: Table View From SQLite 169
Detail View (part of the main.lua file) 172
ZipScreen view (part of main.lua) 174
cityList view (part of main.lua) 176
stateList view (part of main.lua) 178
Summary 180
Assignments 180
Chapter 14: 182
It's Who you Know: Networking 182
Web Services 182
HTTP 182
Project 14: Picture Download – Via Network Library 183
Socket 185
Project 14a: Picture Download – Via Socket Library 185
Tracking Network Status 186
Uploading to a Webserver 187
3-Tier Architecture 187
Post Example 1: Uploading Form Data 188
Post Example 2: Uploading Files or Images 188
Connecting to Propritary Networks 189
Facebook 189
Facebook Example 190
Papaya and OpenFeint 191
Papaya Example 191
OpenFeint Example 191
inMobi 192
inMobi Example 192
Virtual Currency Credits 193
Pubnub 193
Project 14.1 Multi-User App 194

Summary 197
Assignments 197

Chapter 15: 198

Working with Widgets & Popups 198
Widgets 198
Widget Themes 198
widget.newButton 199
widget.newTabBar 201
widget.newTabBar Example 202
widget.newSlider 203
widget.newSlider Example 204
widget.newTableView 205
widget.newTableView Example 206
widget.newScrollView 208
widget.newPickerWheel 209
widget.newPickerWheel Example 211
Removing Widgets 212
Project 15: Longitude and Latitude 212
Web Popups 221
Web Popup Example 222
Summary 223
Assignments 223

Chapter 16: 225

Rotten Apple - a Tower Defense Game 225
Rotten Apples – Inspiration and Resources 225
Adding Sprite Animations 226
I Need a Map! 228
Two Roads Diverged 229
Space, The Final Frontier 230
Rat Race 230
On Your Mark 235
Reducing Collisions 238
Collision Worksheet 239
Take the Shot – Taking Care of Collisions 239
Are We There Yet? – adding the clubhouse 240
Adding Towers: Dragging Towers to the Screen 248
What's the Score? 251
Let's Get this Game Going! 252
Closures 252
Initialization 253
Loop-De-Loop! 255
Level and Wave Control 256
Noises Off! 257
Suspense is Killing Me! - adding suspend/resume/save options 258
It's a Splash - add splash screen 261
Summary 262
Assignments 262

Chapter 17 263

Additional Resources **263**
Autocomplete **263**
BBEdit **264**
Corona Comic **264**
Corona Project Manager **264**
Corona Remote **264**
Crawlspace **265**
Director **265**
Kwik **265**
LevelHelper **265**
Lime **266**
Physics Editor **266**
SpriteHelper **266**
Spriteloq **266**
Texture Packer **267**
Tiled **267**
Useful Websites **267**
Free Isometric images 267
Music 267
Sound effects 267
Tutorials 267

Appendix A **268**

The Lua Language **268**
Lua **268**
An Introduction **268**
What is Lua? **268**
Lua in Practice **269**
Types and Variables **269**
Type Declarations 270
Nil 271
Booleans 271
Numeric Values 271
Numeric Operators 272
Dividing by Zero 272
Strings 273
Quoting Strings 273
Escaping Characters 274
Concatenating Strings 274
Comparing Values **275**
Boolean Operators **276**
The and Operator 276
The or Operator 276
The not Operator 276
Stacking Boolean Operators 277
Lua Data Functions **277**
String Functions 277
Finding the Length of a String 277
Global Substitution 278
Finding a Pattern in a String 279
Matching a Pattern in a String 279

Obtaining a Characters Byte Value ... 279
Getting a String Value from Bytes ... 280
Changing the Case of Characters ... 280
Retrieving a Segment of a String ... 280
Math Functions ... 281
Function ... 281
Parameters ... 281
Returns ... 281
A Note About Code Blocks in Lua ... 282
Conditional Statements ... 283
The if Statement ... 283
Using else ... 283
Nesting if Statements ... 284
Loops ... 285
The for Loop ... 285
The while Loop ... 286
The repeat Loop ... 287
Using break ... 287
Custom Functions ... 288
Defining a Function ... 288
Returning Values from a Function ... 289
Returning Nothing ... 290
Returning Multiple Values ... 290
Multiple Assignment in Variable Definition ... 291
Multiple Assignment from Function Return Values ... 291
Multiple Return Values as Function Parameters ... 292
Value Lists ... 292
Summary ... 293

Appendix B ... 294

Advanced Lua ... 294
Lua ... 294
Advanced Topics ... 294
Understanding Variables ... 294
Global and Local Variables ... 294
Understanding Scope ... 296
Functions and Variable Scope ... 297
Closures ... 298
Garbage Collection ... 299
Functions with Variable Arguments ... 299
The VarArg Operator ... 300
Select ... 301
Recursion ... 302
The Table Type ... 303
Associativity ... 304
Tables as Arrays ... 305
Array Indices ... 305
Creating Arrays ... 305
Arrays are Tables Too! ... 306
Unpacking Arrays ... 307
Finding the Length of an Array ... 308

Looping Over Arrays with ipairs ... 308
Adding Values to Arrays ... 308
Removing Values from Arrays ... 309
Converting Arrays to Strings ... 309
Sorting Arrays ... 310
Finding the Largest Index ... 311
More on Tables ... 312
Iterating Through Table Keys ... 312
The next Function ... 312
The pairs Function ... 313
Object Oriented Programming in Lua ... 313
Creating an Object ... 313
Designing Objects ... 314
The self Property ... 315
Metamethods ... 316
Understanding Metamethods ... 316
Registering Metamethods with setmetatable ... 317
Operator Metamethods ... 317
Operator ... 318
Metamethod Signature ... 318
Description ... 318
Accessing Values with the __index Metamethod ... 319
Assigning Values with the __newindex Metamethod ... 320
Using rawset and rawget ... 321
Creating a Pseudo-Class ... 323
Summary ... 324

About the Author

Brian Gene Burton, Ed.D. is a teacher, author, and game developer. He has created game development degrees at two universities and enjoys researching and playing virtual environments. Brian presents and publishes internationally on his research and enjoys sharing what he has learned about game and mobile development. When not traveling or teaching, he can be found at his home in the Ozark Mountains of Missouri with his beautiful wife of over 25 years, Rosemary.

Dedication:
I dedicate this book to my loving wife whose support and encouragement kept me focused and writing.

A special thank you to my students and the Corona community for their support and requests for specific details and editoral comments that helped so much with the development of this book.

Ch. 6 sounds and music loops were graciously provided by Shaun Reed of Constant Seas. You can check out his band at http://www.constantseas.com

Ch. 5 tileset graphics from Reiner's Tilesets (http://www.reinerstilesets.de) are used with permission.

All other graphics (unless specified) and cover designed by Brandon Burton.

Copyediting and formatting assistance provided by Brianna Burton (http://www.LiteraryDiaries.com)

Foreword

Not long ago, circa early 2008, Walter and I decided to go into the app making business. We would create a series of mobile apps for us to sell via the app store as we saw Apple and its newly announced iPhone as the future of smartphones and app distribution.

We decided to build simple apps at first and then progress to more complicated ones. But as we dove into it, and iPhone being in the inchoate state, we weren't sure if it would take off. Being an Apple product, we knew it was going to sell and create quite a splash, but it was too early to tell. Nokia on the other hand, had a huge market share of the smartphone business but app development and distribution were lagging.

So we took a bet and knew that in order for us to be successful at writing our own apps, we needed to cover more than just one platform. On one hand, we knew iPhone was poised to be a winner, but on the other hand Nokia had quite a grip on the market with their series 60's.

As we forged ahead, and started building a framework that would allow us to cover more than one phone base and quick app production. We looked at several different options, scripting languages, and a slew of technologies that we could leverage and create our own framework for us to use.

From our very own learned experience, we knew we had to move fast, and after digesting all the options we had, we settled on Lua and started working on the framework now known as Corona.

Internally, we called this nascent framework "Ratatouille", the name was apropos because it took us back to the days of programming within a constrained memory model, small disk sizes as well as small screen sizes.

After the initial scaffolding, we started building some prototypes of the kinds of things we could do and it was, at that time for us, a glorious moment, we honestly thought we were a bit ahead of our time and we weren't even sure if it was going to work - typical engineering mentality. You work hard and after weeks all that we could show for was a rectangle being drawn on the screen. But the 'aha' moment came when the same code based allowed for the app to work on the Series60 as well as on the iPhone. That quickly removed all doubt and we started adding features to Ratatouille left and right.

At one point, we had about six prototypes we had built and they all worked flawlessly, it was easy to prototype apps with this pre-alpha version of Ratatouille.

Eventually, we dropped support for Nokia and started support on Android, and decided to go knock on some VC's and see if we could make this into a business as we saw quite an interest from our own friends and friends of friends on our product.

Fast forward to today. In typical Silicon Valley fashion, Corona was born from an idea out of our garages in order to solve our needs. Little did we know we would create a tool that would enable thousands of developers to fulfill their entrepreneurial spirit and start businesses using Corona SDK.

Today, over 20,000,000 people have played with Corona-based games and apps. These games/apps are being written all the way from 14 year olds, to teams of dedicated gamers and by ad agencies and studios. And the best part is, there is no slowdown in sight.

But Corona can't just be successful by your apps alone. It also takes time and dedication and learning from trusted individuals like Dr. Burton, who has time and time again created some great tutorials on how to use Corona. And in his own entrepreneurial spirit, he has taken valuable time from his busy schedule to write a book on Corona.

This book is an excellent way to introduce you to our Corona SDK and will serve as the definite go to guide on how to learn and build Corona apps.

I know you will enjoy the book as much as you will enjoy building apps with Corona.

Carlos M. Icaza
Co-founder, Ansca, Inc.

Preface

Welcome

Welcome to mobile application development with Corona. This book is the result of years of developing for mobile devices. In early 2010 I began looking for a better way. I wanted a tool or set of tools that would allow me to develop more quickly and easily for multiple platforms of mobile devices. I was tired and frustrated with having to re-work everything to be able to make the same app on an iPhone, iPad, or Android device. After trying several different tools and development environments, I came across the Corona SDK by Ansca Mobile (http://www.anscamobile.com). While it was still early in the development of the SDK, it was apparent that the Ansca team was committed to building a quality set of tools and that a devoted community was quickly forming around this great SDK (Software Developers Kit). In the early days of my learning Corona, I focused on creating tutorials (available on my website: http://www.BurtonsMediaGroup.com/blog). After teaching Corona to several of my mobile and game development classes, and with the encouragement of my students, I began the process of creating a book that could be used as a teaching resource for the Corona SDK. You hold the fruits of that endeavor in your hand. I hope that you enjoy learning Corona as much as I have!
Best wishes,

Brian G. Burton, Ed.D.

Who This Book Is For

While my focus and impetus for writing this book is that it be used as a textbook, I have also written it with the understanding that many (hopefully) are just interested in learning more about the Corona SDK and want to develop for multiple mobile devices at the same time. I have the expectation that anyone using this resource already has some basic programming knowledge and experience. I do not spend very much time going over programming fundamentals. There are many great books on programming, I recommend you start there and return to app development when you have the basics.

How This Book Is Organized

While writing this book, I have kept the traditional 16-week semester in mind, assuming one chapter per week. While that doesn't work for everyone, it should be enough for most people to get started with mobile development using the Corona SDK. My first draft ended up with more than 20 chapters. After reorganizing content and continuing to develop, we are now down to 16 chapters with an additional chapter on great resources and a couple of appendices that were graciously supplied by Ansca Mobile on the Lua scripting language.

Conventions Used In This Book

Throughout the book I will use `Courier New` font to denote code that should be typed in exactly. When you find examples that are in *Courier New, Italics* you will need to enter your own value.

Using Code Examples

This book was written to help you learn to develop applications and games with the Corona SDK. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission for reproducing a significant portion of the code. You don't need to ask permission to write an app that uses large chunks of code.

Now, on the other extreme, if I see apps that exactly reproduce the examples from a book or tutorial, I will not be a happy camper. I don't have issues with using the examples as a starting point, but take the app much further; be original! Answering questions by citing this book or quoting examples does not require permission (but I would appreciate the citation).

I reserve all rights for selling or distributing the examples in any format provided in this book. If you're not sure if your use falls outside of the fair use laws, please feel free to contact me at: DrBurton@BurtonsMediaGroup.com

Why didn't I use _______ for _______

There are a lot of great products available that can help the budding programmer/developer get their work done much faster (see chapter 17 for a short list). As this book is aimed at college students and people just getting started, I tried to not use outside tools. If a tool was required to get the project done, I tried to use only free or low cost tools. If I didn't use one of your favorites, I either 1) didn't know the tool existed; 2) was unable to get an evaluation copy of the software in a timely fashion; or 3) just didn't like that tool (probably the first or second option). If you know of a great tool that can save time and money to developers, please share it with the world in the discussion board on this books site: http://www.BurtonsMediaGroup.com/books.

Appendices

Appendices A and B on the Lua programming language were supplied by Ansca Mobile and are included with permission. While we have performed some copy editing to (hopefully) improve readability, the original content and examples have remained as provided.

How to Contact Us

Please address any comments or questions to the books website: http://www.BurtonsMediaGroup.com/books or email DrBurton@BurtonsMediaGroup.com.

Why I Chose to Self-Publish

The decision to self-publish this book was reached after a great deal of consideration. While there were numerous publishers interested (both academic and technical), I have decided to publish at least this first edition without the use of traditional publishers. There are many reasons why I made this decision, even though it will most likely lead to fewer sells.

First among my concerns was the price of the final book. I am sick of seeing textbooks at $100+. I feel this pricing is wrong and places an undue burden upon students. While publishers have cut the price slightly with the advent of eBooks and eTextbooks, it hasn't been enough in my opinion.

My second concern was how rapidly software environments change. I personally hate having to purchase a new book for each major revision of software. I have stacks of books that are now completely useless. I decided to publish this first as an eBook, which allows me to update and provide it to you, the reader, more rapidly. I will provide the minor updates between editions to the eBook to everyone who purchases the eBook through my website: http://www.BurtonsMediaGroup.com/books/book-update/

That being said, if you received a copy of this book either through a torrent or a friend, please purchase your own copy through my website. This will provide you with the most recent version of the eBook and encourage me to continue to update it. While I am doing this to help my students, I have bills to pay, and my wife is really good at keeping my 'honey-do' list up-to-date. Help me to avoid that list by buying a legitimate copy of this book (I don't have to work on her list if I'm writing or editing).

On the downside of self-publishing, I do NOT have a team of people to proof and double check everything in this book. I am sure that typos were entered by gremlins during the night. That and I have dyslexia. I did hire a person to proof the final version of the book, but having read many books that were published by major companies and finding errors in their books, I am sure that errors remain in this one. Please let me know if you find a typo on the book's forum site: http://www.burtonsmediagroup.com/forum

Chapter 1
Hello World: Setup and Get Going!

You've got a great app/mobile game idea. Wonderful! Now, how do you create it and get it on to an iPhone/Droid/iPad, (or whatever your device of choice is)? There are so many devices to choose from. Which platform is best for my app?

With so many platforms to choose from (Android, iOS, RIM, Windows, to name a few), the choice of platform to develop for can be very difficult. Each platform uses a different language, has a different API (Application Programming Interface) and requirements. How willing should we be to get locked into one development platform? Should we choose just one?

Fortunately with the advent of tools such as Corona by Ansca Mobile, it is now possible to develop for multiple platforms at the same time. To write once and publish to a host of different devices is the ultimate solution in the mobile publishing world.

Ansca Mobile's Corona currently allows the budding developer to publish to Android and iOS (Apple) devices, be it a smart phone or tablet. This text is written to help students everywhere gain the fundamental skill set to be able to take their app idea and publish it using the Corona SDK.

What this book is not

While this book is designed to teach the basic of mobile application development, it is not designed to teach programming fundamentals. I am making the assumption that you already know the basics of computer programming. If you don't know how to use an "if then" statement, a loop or a function, you're probably not ready for this book.

While I have made every attempt to cover the basics that most students want to learn during a 1st semester course in mobile app development, due to space and time issues, only so much could be included. There is already a second volume in development that will cover more advanced mobile application development and a volume that is just on game design with Corona.

Getting Started

Corona: Some background

Ansca Mobile was created in 2008 as a venture-backed company in Palo Alto, California. Before Corona, the Ansca Mobile team was responsible for creating many of the industry standard tools that I am sure you are familiar with. In the time that I have been developing apps with Corona, I have found Ansca Mobile to be one of the friendliest and helpful businesses that I have had the pleasure of working. In addition, online community is

unusually friendly and supportive. If you decide to join the Corona community, be sure to continue this great spirit of helpfulness!

Software: Corona

It's no surprise that you will need the Corona SDK to get started. For learning, I recommend downloading the trial version. If you are ready to become a full subscriber, just head over to the on the Ansca Mobile website http://www.anscamobile.com/. Click on the download button and register (whether you are purchasing the subscription or downloading the trial). If you are a student or faculty, you can get a discount on your subscription by going to http://developer.anscamobile.com/forms/educators-and-students.

Software: Android

To get started developing apps for Android devices with Corona, you do not need to download any additional android software. However, you will need the Java SDK (typically referred to as JDK) to be able to do device builds if you are on a Microsoft Windows system. Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html and download the Java Platform Standard Edition 32 bit JDK 6 by clicking on the "Download JDK" button. Note: You only need to download the JDK if you are on a Microsoft Windows system. If you are using a Macintosh with OS X, it is already installed.

Examples and Graphics

One last download that you can take care of right now: the code examples, graphics and other tools that you might want to use with the projects that are listed in this book. They are all available at http://www.BurtonsMediaGroup.com/books.

Software: iOS

For straight app development on the simulator, you don't have to download anything from Apple. However, when it is time to deploy to your test device or prepare the app for the iTunes store, you will need the Apple iOS SDK. If you are already an Apple developer, then you should be ready to deploy. If you are not a current developer ($99 per year for a standard iOS developer), you can register for a free developer account and download Xcode through the Apple App store for $4.99).

Development Hardware: Corona

Corona isn't too demanding on your development computer. As long as you are running at least OSX 10.6 or later on the Mac side, or Windows XP with a 1 GHZ processor on the PC side, you will be fine.

If you are planning to develop and deploy to iPhone, iPod Touch, and/or iPad, then you will need a Mac of some type to develop your apps. This is an Apple requirement. To keep in everyone's good graces, Corona will only publish for an iOS device if you are using a Mac computer to deploy the app. You will also be able to develop and deploy your Android based app from a Mac.

If you only have a windows system, you will be able to develop and deploy for Android based devices. You will also be able to develop for iOS devices. You just cannot deploy your finished app to an iOS device (or the iTunes store). I use both a Mac laptop and a PC, regularly switching back and forth during the app development process.

Development Hardware Matrix:

Development Hardware	Android OS		Apple iOS	
	Develop	Deploy	Develop	Deploy
Macintosh	X	X	X	X
Windows PC	X	X	X	

Development Hardware: Test Devices

If you are going to develop and sell apps for mobile devices, you should have a mobile device to test your creation. I have been on projects where I was required to develop for hardware that I didn't have. It was like herding cats. Using just the app simulator will get you 75% of the way home, but it won't allow you to spot all potential problems. On one of the fore-mentioned projects, the app worked fine on the simulator, but crashed on the mobile device and was rejected by Apple. The experience was more than just a little frustrating and taught me a valuable lesson: If you are developing for a platform, have test devices!

Android

Corona only builds for Android OS 2.2 and newer. Any devices that you plan to develop for must use the ARM V7 processor. There are plenty of devices that meet this requirement, so you shouldn't have any problem finding one to perform your tests.

iOS

For developing on iOS, you will need a developers license and either an iPhone, iPod Touch, or iPad. Obviously, having an older phone or iPad is a good idea for testing FPS (Frames Per Second) for graphically intensive apps. It is recommended that you use the newer iOS on your devices. To be able to deploy to an iOS device, you will need a Mac computer system and a Standard, Enterprise, or University developers account from Apple.

Publishing Considerations

I am sure that you are already envisioning how you will spend that first big check from your app sales. But before you can sell your app, you will need to decide with whom you will publish your app. There are several considerations specific to each publisher that you need to keep in mind.

Android (Google)

The Android market is very different from the Apple iTunes store. With the Android app market you have a number of different vendors available for selling your apps, including the Google's Android market (http://market.android.com), Amazon, Barnes and Noble, and a host of vendors. You will need to create an account with each vendor that you wish to sell through.

For $25 dollars (US) you can setup a developer account for Android with Google. You do NOT have to sign up for any account until you are ready to begin selling your apps. To get started visit setting up your personal account, visit http://developer.android.com/index.html.

Once you have your account setup you will need to decide if your app will be free or if you will charge for it. Throughout most of the mobile app industry the split is 70/30 in your favor. In other words, if you charge 99 cents for an app, you will walk away with 69.3 cents on each sell.

iOS (Apple)

One of the biggest advantages of the iOS market is that there is just one market to belong. To develop apps for the iOS market it costs $99 per year for a standard developer's license. An enterprise developer's license is also available, but unless you are developing for a major company that will only deploy your app internally, you will want the standard license. On a student budget, $99 can seem pretty expensive, so I recommend waiting as long as you can before getting your standard license since it is only good for one year. You can explore the developer's license options and the iOS SDK at: http://developer.apple.com/programs/register/. Apple also follows the 70/30 split on app sales.

Programming in Corona: Lua

In this text, the language that you will use throughout your programming experiences with Corona is Lua. Lua is a scripting language that was developed in the early 1990's. It is free, distributed under the MIT license and widely used for level scripting in major games and is a natural fit to be used in mobile application development due to the small size of the interpreter. Ansca Mobile has been kind enough to provide the two appendices on Lua (Appendix A & B) that are included in this book. If you would like to learn more about the Lua language, you can visit the Lua home page at http://www.lua.org. The first edition of

Programming in Lua is available online at http://www.lua.org/pil. If you have ever programmed or scripted in any modern programming language, you should find Lua to be easy to learn as we progress through the following lessons.

Programming in Corona: Editors

The editor that you decide to use is a personal decision. Corona isn't impacted by the editor selection, so you need to use an editor that you are comfortable with. I recommend one that allows the integration of Lua to make your editing easier.

Some of the most popular editors in use with Corona include (but are not limited to) BBEdit, Eclipse, Notepad++, TextMate, TextWrangler, and Xcode. Of course you can ignore all of these editors and use notepad or textedit if you so desire.

BBEdit (Mac) by Bare Bones software, $99.99.
I have been using BBEdit on my Mac for quite a while and it is my editor of choice when working on my Mac. BBEdit has built in configurations (including Lua), which easily allows you to set the editor to the language you are developing in. http://www.barebones.com

Corona Project Manager (Mac/Win) by J.A. Whye, $75.
Corona Project Mangaer has a built in editor. Coupled with its ability to greatly simplify tracking your Corona project, the cost of CPM is well worth it. See Chapter 17 for a coupon code to save 30% on CPM. http://www.coronaprojectmanager.com

Eclipse (Mac/Win) Open source, $0.
Eclipse is the editor I use when working on my PC. Eclipse has a large community of support. Though Eclipse was originally designed as a Java IDE (Integrated Development Environment), it is now the bases for many editors on the market. A Lua/Corona plugin is available. http://eclipse.org

Notepad++ (Win) Open source, $0
A popular open source language editor for the PC environment. http://notepad-plus-plus.org/

TextMate (Mac) by Micromates, €39 (about $57).
Textmate is very popular in the Corona community with a Corona plugin available on the Ansca Mobile website. http://macromates.com

TextWrangler (Mac) by Bare Bones Software, $0.
TextWrangler has the advantage of being a free editor for your Mac. Though it doesn't have all the bells and whistles as BBEdit, it will get the job done for those on a budget and offers integrated Lua support. http://www.barebones.com

Xcode (Mac) by Apple, $0*.
Xcode is an integral part of the iOS SDK. If you are used to developing using Objective-C, Xcode is a natural choice. While Xcode is included with iOS SDK, it is only free if you are

already a standard developer with Apple. If you register for a free account, the iOS SDK (which includes Xcode) is $4.99.

Configuring Corona

Installation of Corona SDK is a straightforward project. Just click on the download button at http://www.anscamobile.com, register, select whether you are downloading the Mac OS X or Microsoft Windows version of the Corona SDK, and follow the directions below based upon your operating system (images may vary depending on the version you are installing).

Corona SDK Download

Macintosh

After you launch the downloaded file and agree to the software license, drag the Corona SDK folder onto the Applications folder.

Installing Corona SDK on a Macintosh

This will copy all of the Corona SDK files in to your applications folder. When you open up your Corona SDK folder, you will find sample code, tools, a resource library as well as the Corona Terminal and Simulator (the primary development tools that we will be using).

Windows

Corona SDK for Windows has low hardware requirements:

- Windows 7, Vista, or XP operating system
- 1 GHZ processor (recommended)
- 38 MB of disk space (minimum)
- 1 GB of RAM (recommended)
- OpenGL 1.3 or higher graphics system

In all of the installs that I have made of Corona, the only problem I have ever run into was when a system didn't have OpenGL 1.3 or higher. This was easily corrected by downloading newer graphics card drivers to the system. Corona SDK will run with older versions of OpenGL installed, as long as it is an application that is graphic intensive. You should be able to update your graphics card driver to correct the problem if it exists. More information about OpenGL can be found at http://www.opengl.org.

If you haven't already downloaded the Java JDK (Java Developers Kit), you should do so now. Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html and download the Java Platform Standard Edition by clicking on the "Download JDK" button. On the next page, select "Windows x86" from the list of available downloads.
The JDK is required to be able to do device builds on Microsoft Windows systems. This is a free download from the Oracle website. After you have downloaded the installer, follow the normal procedure to install the JDK to your system.

Programming in Corona: Hello World (V1.0)

The first time you launch the Corona Terminal or Simulator it will ask you to login with your registration information that you used on the Ansca Mobile website. Complete this one time authentication and you will be ready to go.

Corona Developer Registration

You should always launch the Corona Debugger on a Macintosh instead of the Simulator for performing application builds and testing. On a Windows system, launching the Corona Simulator also launches the Corona Simulator Output window (commonly referred to as the terminal window). The Corona Terminal gives you important feedback when you are building your apps and allows for easier troubleshooting. The Corona Terminal will automatically launch the Corona Simulator.

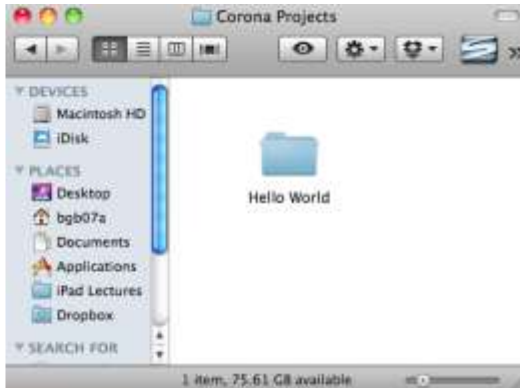
Project 1.0: Hello World

I personally always hated programming books and classes that spent the first chapter or week just getting all the details taken care of. I purchased the book or took the class because I wanted to program, not to go over some syllabus or a review of all the different ages of computer development. So let's skip all of that and do the required "Hello World" project.
Stop with the rolling of eyes! Before I lose you, let me guarantee that you will get a very valuable resource out of this Hello World project, something that you will use the rest of the time that you develop in Corona.

Was that enough to get your attention? Then let's get started!

Project Setup

If you follow this process each time you start a new project, it will make your life a lot easier:
First, create a project folder called "Hello World". This can be on your desktop or wherever you like to organize your work. I keep all of my project folders together in a folder called "Corona Projects".



Create the Hello World folder for your project

Open your editor of choice (I'm using BBEdit in these initial screen shots). Create a blank file and save it as "main.lua" to your Hello World folder that you just created. The main.lua file is the first file that the Corona simulator will look for when it is run. If there is no main.lua file present, nothing will happen.

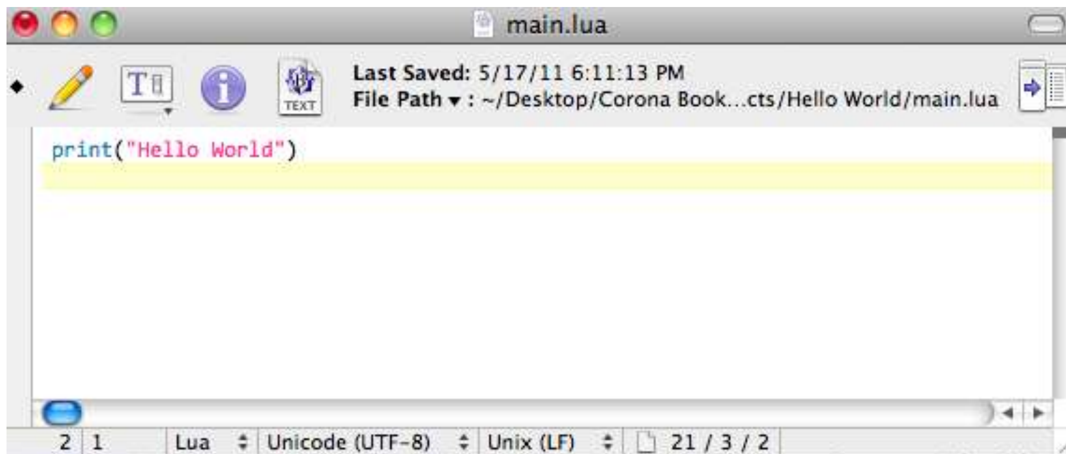
Save the main.lua file to your Hello World folder

There should now be a main.lua file in your Hello World folder.

Back in your editor type :

```
print("Hello World")
```

and save your file as main.lua.

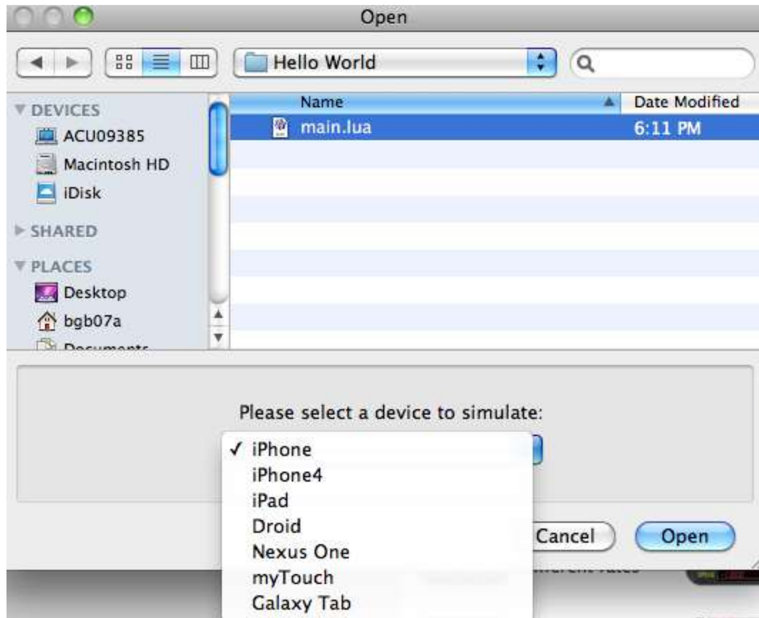


Hello World project in the editor

Next, you will need to launch Corona. If you are on a Microsoft Windows system, launch the Corona Simulator. On a Macintosh, launch Corona Terminal.

Corona at startup on a Macintosh – don't use the new project button yet!

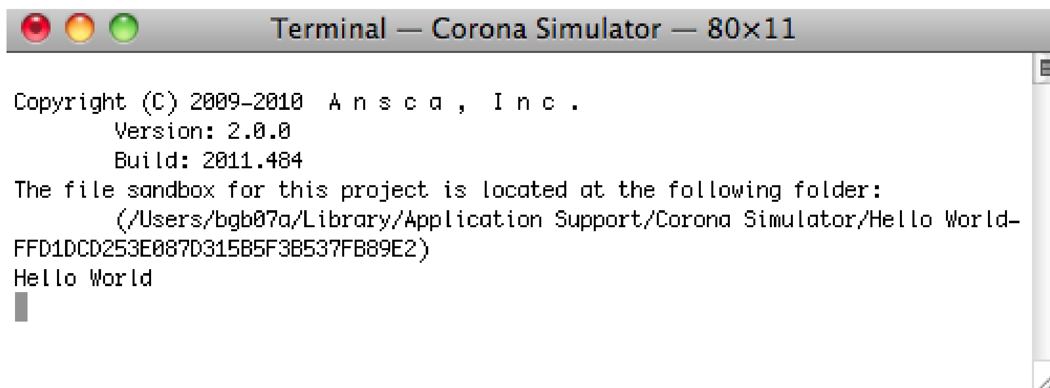
On launch, you will see the Terminal window and the Welcome to Corona dialog box. Select "Open a Project" from the Welcome to Corona dialog and navigate to the Hello World folder that was created earlier. Your initial window might be different based upon the version of Corona that you are using.



Open Hello World & select device to simulate

When you are opening a project, you will be able to select which device you would like to simulate in the Corona Simulator. For now select either iPhone or Droid and click on the Open button. Selecting other devices could give you different results than what are in the screen shots that have been included.

As soon as you open the project, the simulator will run the project.
Did you notice? That's right, nothing happened…in the simulator. Look in the Terminal window.
At the bottom you will see your Hello World displayed.



Hello World in the Corona Terminal window

Congratulations! You just made your first Corona app! Now before you become disappointed, you just learned a very important tool for trouble shooting your applications. When something doesn't seem to be working correctly or displaying the way you want, you can send yourself messages through the Corona Terminal window. Believe me when I tell you that this one command will save you hours of troubleshooting headaches!

I am sure you also noticed that Corona generates a great deal of additional information before giving you the results of your print command. The first few lines provide information about the version of Corona and the location of the simulation files.

Note: If you didn't see anything, there are two areas that people commonly make a mistake: 1) they didn't save their main.lua file (I still make this mistake) or 2) when saving the main.lua file, it wasn't saved as a text file.

Project 1.1: Hello World (v2.0)

Back in your editor (you can use the same file)
Type:

```
local textobj = display.newText("Hello World", 50, 50,
native.systemFont, 24)
textobj:setTextColor(255, 255, 255)
```

Lua, the language behind Corona, is case sensitive so newText is different from newtext. Try newtext and look at the error that appears in the Terminal window.
Save the file, and then launch your simulator.

You should now see Hello World displayed in the simulator.

Hello World on the Droid simulator

What did you just do? Here's the run down:
First we created a local variable called textobj. We do not have to use the variable name textobj, we could use fred for the variable name but after a couple of days we might forget what fred represents. Use variable names that make sense. It might mean more typing, but you will really appreciate it when you go to revise or update the program at a later date.

We set textobj equal to the object that we create by calling display.newText method, passing it the text "Hello World", the X & Y location of the top left corner of the text, font, and 24 (the size of the text to be created).

The display.newText parameters are:

```
display.newText(text, X, Y, font, text size)
```

In the font parameter, you can use the system default of native.systemFont or native.systemFontBold. You can also enter the font type in as a string such as "arial" or "arial black". If you set this parameter to *nil*, it will default to the native.systemFont.

In the second command line, we set the color of the textobj that was just created using the R, G, B color system (each color (red, green, blue) having a value between 0 – 255) to white:

```
textobj:setTextColor(R, G, B)
```

By default, the text object is white, so we didn't really accomplish anything by setting the textobj to white. I want to get you in the practice of setting the text color when you create a text object. Later we will look at how to fade the text object out (or in).

Now you have made your first REAL Corona app!

Warning: If you copy code from a website (or even from this book), sometimes the quotation marks will change from straight quotation marks to smart quotes. This WILL cause an error in Corona. Make sure your quotes are always `""` and not "".

Objects

You may have noticed the use of the term object sprinkled throughout the text thus far. When I use the term 'object' it is to denote anything that is used in our project; text, buttons, or sounds, they are all objects. Just as in the real, physical world, I can move or interact with an object (a lamp, table, or car), an object in your software is anything that can be interacted with.

Real world objects all have properties that help to describe the object's location, color, or anything that can be changed about the object. If I have a car, I might describe the car's location by its longitude and latitude.
In programming (including Corona), we are able to interact with each objects properties to make changes; such as when the textobj was created, we set the x, y, font, and size properties as well as the string that would be displayed.

Summary

This has been a busy chapter! Corona should now be installed on your system, you have been introduced to editors, hardware considerations, and publishing information. We even managed to develop two apps! The first introducing the critically important print command, the second actually displaying text to the simulator. Finally, the concept of an object in programming was briefly introduced.

Assignments

1. Try various typos to see the resulting error messages in the terminial window.
    a. Make a typo in newText. What is the result?
    b. Make a typo in native.systemFont. What is the result?
    c. Try settextColor. What is the result?
2. Change the text object to red.
3. Reposition the text to the bottom of the simulator without letters going off the bottom.
4. Place 5 different messages in different places on the screen, each in a different font, size, and color.

Chapter 2
Buttons and Text

We are going to combine a few things with this Chapter:

- First, we will learn about creating an object that is interactive
- Second, we examine how to set or change the location of an object
- Third, we will add an event listener to the object so that we can interact with the app
- Fourth, we will make use of the math.random function to help us move an object around the screen.
- Finally, we will look at the Relaunch feature of Corona Simulator, which makes it so easy to tweak your code!

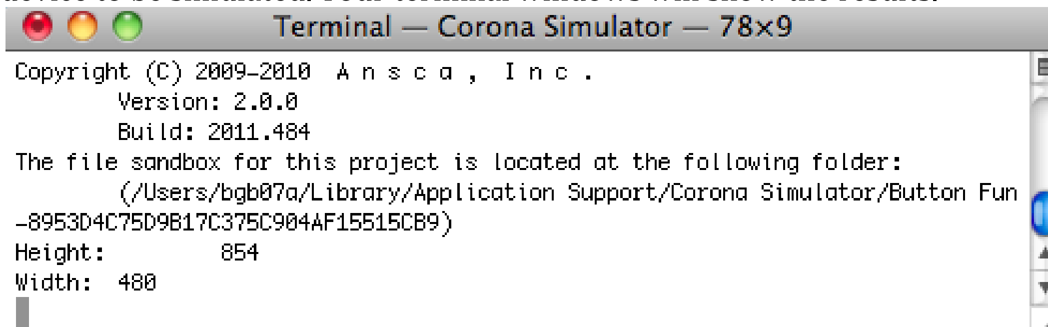
Know your Boundaries

When developing for multiple types of devices it is important to automate the placement of objects on the screen. In Corona, this can be done easily with display.contentWidth and display.contentHeight.

For our second project, let's start by finding the height and width of your device. This can easily be accomplished using the print command that was discussed in chapter 1. Start by creating new folder and main.lua for your app. Add the following code:

```
print("Height: ", display.contentHeight)
print("Width: ", display.contentWidth)
```

Launch Corona as you did in the previous chapter (Corona Terminal for Mac, Corona Simulator for Windows), open main.lua in the Button Fun folder, then select the type of device to be simulated. Your terminal windows will show the results.



Display Content Height and Width

In this example I selected Droid for my simulator. As you can see in Figure 1, the height is 854 pixels and width is 480 pixels.

Project 2: Button Fun

For this project we are going to create an app that will move a text object to a random place on the screen each time the button is tapped. For this project you will need to create a button graphic. I just went into Photoshop (or gimp, paint, or any other graphics software) and created a small 100 pixel by 50 pixel rectangle and saved it as button.png. Make sure you save it to the same folder as your main.lua file.

To load the button.png into our app, we need to create an object to refer to the graphic:

```
local myButton = display.newImage( "button.png" )
```

This line of code creates a local variable called myButton, then assigns an image to it (the button.png graphic we just created).
If you save the main.lua file and run the simulator, you should see the button you created located in the top left corner of the simulator. You could also set the top left corner location of the graphic by adding the left and top as we did with the textobj in chapter 1:

```
local myButton = display.newImage( "button.png", 100, 100 )
```

Instead, we will change the x and y location of the myButton object directly by changing the property setting. One important difference between setting the location in the display.newImage and changing the x & y values. In the display.newImage, you are setting the left and top location of the object. When you set the x & y values, you are setting where the center of the object will be located. Try entering the following code after you create your myButton object and see the difference:

```
myButton.x = 100
myButton.y = 100
```

Great! We are able to move the myButton object anywhere on the screen we want. But there is a problem. There are a lot of different types of devices that we can build for with Corona and each one has a different resolution. Wouldn't it be nice to have it located in about the same place on the screen no matter what type of device it is running on? Fortunately this is easy with the commands we have already played with: display.contentHeight and display.contentWidth!

Using a little simple math, we can place the myButton object in the exact center of the screen. Replace the original myButton.x and myButton.y with:

```
myButton.x = display.contentWidth /2
myButton.y = display.contentHeight /2
```

and save, then run your app. The button should now be in the exact center of your screen.

Button is now in the center of the screen

Let's adjust the button a little more. For the next part of my project, I want text to be displayed when I press the button. To simplify the interface, I want the button at the bottom of the screen. Again, with a little math, this is easily accomplished. Since we know that the button is 50 pixels in height, that the y property looks at the center of the object, and I know the height of the device in pixels from the variable display.contentHeight, we can easily place the button 50 pixels above the bottom of the screen with: myButton.y=display.contentHeight – 75 (50 pixels from the bottom + 25 pixels for the center of the object). Replace the myButton.y = display.contentHeight/2 with:

```
myButton.y=display.contentHeight - 75
```

It is important at this point to consider the esthetics of the app in the sense of the button size. Too small a button and users finger might be to big, too big a button and you will waste limited screen space. Examine some buttons from mobile apps, iOS, Windows phones, etc to get a concept of what is the right button size for your app. Remember, once you have created a button you like you can reuse it for other projects.

Functions

Now we need to tell the button what to 'do' when it is tapped. This will be done using a function. A function is a group of commands that will only be executed when called. Whenever there is a need to perform the same operation multiple times or only after specific events (such as a tap on a button), functions provide this ability.

A function is begun with the keyword 'function' (surprised?) followed by the name of the function and any parameters to be passed. A function always ends with the keyword 'end'.

In between these two keywords will be the commands and operations that you want to be executed when the function is called.

For our first function we are going to write some code that will randomly move the text each time the myButton is tapped. Before we write that function we will need to go back and add the text object that will be moved. Add:

```
local textobj = display.newText("Button Tapped", 10, 50,
native.systemFont, 24)
textobj:setTextColor(255,255,255)
```

below myButton.y= display.contentHeight-75

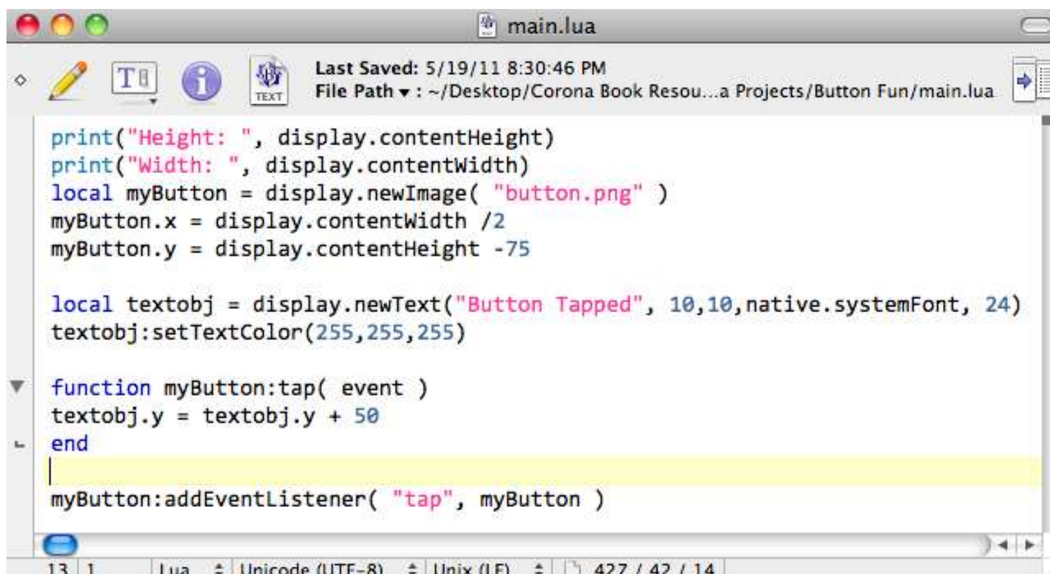
Now for the function. To begin with, we are going to have the text object (textobj) move down a few pixels every time the button is tapped.

```
function myButton:tap( event )
textobj.y = textobj.y + 50
end
```

In this function, we are moving the textobj down 50 pixels by retrieving its current Y position and adding 50 to it until it eventually moves off the bottom of the screen.
One last line is required before we test our app:

```
myButton:addEventListener( "tap", myButton )
```

Add this command as the final line in your code. This sets up an event listener (but I'm sure you guessed that from the name of the command) that listens for a tap event to occur on myButton.



Save your project and run it in Corona. With your mouse you can click on the button at the bottom of your screen, which simulates a 'tap'.

Hmm, it seems we have a problem. After several taps, the text object will move beyond the bottom of the screen. Let's add an 'if then' statement to the myButton function to catch this problem.

```
function myButton:tap( event )
    if (textobj.y > display.contentHeight -120) then
     textobj.y = 30
    else
     textobj.y = textobj.y + 50
    end
end
```

By using the if then decision statement, we can check to see if the object is below the button location and move it back to the top if it is below the button.

Project 2.1: Button Fun V2

Let's make this project a little more interesting. Using a random number generator, we can relocate the text object to a new location with little effort, and make the project more interesting at the same time. The random number generator in Corona is part of the math command set and is called math.random(low, high). Since we are building for a variety of devices, we will use display.contentWidth and display.contentHeight for our high values. By adding these two lines of code to our function and removing the if then statement, we can now relocate the text object to a new, random, location.
Your myButton:tap function should now look like:

```
function myButton:tap( event )
   textobj.x = math.random( 0, display.contentWidth)
   textobj.y = math.random( 0, display.contentHeight)
end
```

Note: Corona is case sensitive. If you are getting errors, it is probably caused by a typo in either a variable name or a command name.

Tip: When making major changes to your code, it is often easier to just comment out the line of code that you don't want rather than deleting it. Comments in Corona are noted by placing a double hyphen -- at the beginning of the comment. You can begin a comment at any point on the command line. To comment out blocks of code, use --[[ ]].

Save and try it in your Corona Simulator.

Getting Fancy!

Did you notice that sometimes the text object (your "Button Tapped") goes off the screen? That is because the .x and .y properties are setting the location based upon the center of the textobj. The program can't tell where the edges are on the object. For all it knows, we WANT only part of the object to be showing!

There are many ways to keep this from happening. One method is to modify the .x and .y calculations so that the number returned doesn't allow the text to be cut off. Using trial and error, we can adjust the numbers until finally we get:

```
textobj.x = math.random( 85, display.contentWidth -85)
textobj.y = math.random( 20, display.contentHeight - 110)
```

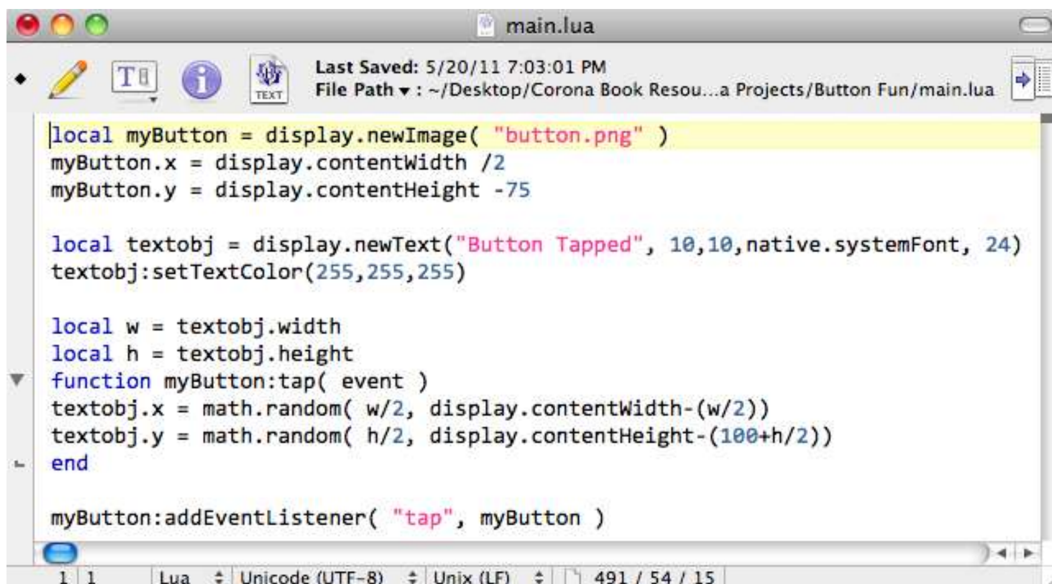
In this case we are generating a number between 85 and the content width -85 for x and a number between 20 and the content height -100 for y. I chose the 100 pixels value so that the text object is always above the button. This pretty much keeps the text on the screen at all times. You can make these changes, save, and then click on the simulator, File > **Relaunch**. Relaunch reloads your main.lua with the changes you made. It saves you from having to do it with Open each time (a wonderful feature when you're trying to trouble shoot a project).

A better (and there are other even better methods, but this will do for now) method is to look at the size of the object that you want to keep on the screen. Since you might not know the size of the object when the program is running (for a variety of reasons), it is better to let the program figure out what will keep the object fully on the screen:

```
local w = textobj.width
local h = textobj.height

function myButton:tap( event )
textobj.x = math.random( w/2, display.contentWidth - (w/2))
textobj.y = math.random( h/2, display.contentHeight - (100 +
h/2))
end
```

The .width and .height properties return the size of the object in pixels. This makes it easy to calculate where the object can be placed on the screen. Of course, we don't have to set the text object width and height to a variable, but it does make the random number calculation a little easier to read.

By creating two variables (h & w), we are recording the height and width of textobj. This can then be used in creating a simple formula to keep our text on the screen!



Now textobj.x is limited to a number between half the width of the text object and the content width minus half the width of the text object. Similarly, textobj.y is limited to a number between half the height of the text object and the content height minus half the height of the text object and the 100 pixels.

How Corona reads your main.lua file

Now that you have been introduced to functions, you might be wondering how Corona processes the main.lua file. Corona processes your file from top to bottom, one time, just like most applications, unless you are using a loop to continue to make function calls. Corona will continue to listen for any event that you have included, so the app will continue to function until it is shut down.
This is why you will usually load any variables and outside files (we will get to that soon) at the beginning of the file, then declare your functions, and finally make any needed function calls and add event listeners.

Summary

In chapter two we have added to our knowledge base the ability to load graphics, move them around the screen, and turn them into buttons. We learned how to generate a random number, return the width and height of an object, and implement an event listener. Also thrown in just to move things along was an introduction to commenting and functions.

Assignments

1. Create an application with two buttons; one red, one green. Tapping on the red button places the word “Red” at a random location on the screen. Tapping the green button places the word “Green” randomly on the screen.
2. Create an application that keeps track of how many times the button is tapped and displays a running total on the screen.
3. Create 10 number buttons (0 thru 9) similar to what you would find on an inexpensive calculator. Write an app that, when a number button is tapped, the corresponding number appears near the top of the screen. The output font should be fairly large. Make the number buttons small enough that another row of buttons can be placed along the right. We are eventually going to build a simple calculator with these buttons.
4. Create an application that displays the height and width of the display to the screen. Try the app in several different device views (in the simulator: View > View As) and record the display sizes of 3 different devices.

Chapter 3
Animation, Alpha & Orientation

In this chapter we are going to examine three important concepts for interacting with a mobile environment.

- First, we will examine a couple of ways to accomplish basic animation with Corona.
- Second, using the alpha setting, we see how to fade objects in and out of view.
- Finally, we will examine the all-important orientation change settings, critical to many apps (and getting approved by Apple and Google).

Animation

We have already learned how to move an object to a new location on the screen in the previous chapter. There are many ways that animation can be achieved on a mobile device. In this section we are going to look at two ways to animate an object.

Project 3: Basic Animation

In this first project we are going to use a traditional loop to move a square graphic toward the center of the mobile screen. For this example, you will need to create two graphics, a small square (50 x 50 pixels should be about right), and just to make it more visually interesting, a graduated white spot that we will place in the center of the screen. Of course, both of these should be in the png format for greatest compatibility between devices. (Images and other code samples can be downloaded from http://www.BurtonsMediaGroup.com/books).

Create a new folder for this project and copy your images into the folder.
Open your editor and save a file to your folder as main.lua.

To begin, first load first the graduated white dot and then the square into your app. Place the graduated white dot in the center of your display, then place the square at a random location on the screen.

```
--Load images into memory and store in local variables
```

```
local center = display.newImage("Ch3Center.png")
local square = display.newImage("Ch3Square.png")

-- place the center graphic in the middle of the display
center.x = display.contentWidth/2
center.y = display.contentHeight/2

-- place the square at a random location on the screen, but not
off the screen
square.x= math.random(square.width/2, display.contentWidth -
square.width/2)
square.y= math.random(square.height/2, display.contentHeight -
square.height/2)
```

Now for the fun part. We are going to move the square toward the image with a while loop and a couple of if then statements.

```
while (square.x ~= center.x or square.y ~= center.y) do
     if (square.x > center.x) then
          square.x = square.x -1
      elseif (square.x < center.x)then
          square.x = square.x +1
     end

     if (square.y > center.y) then
          square.y = square.y -1
      elseif (square.y < center.y)then
          square.y = square.y +1
     end
end
```

With this while loop, we are telling the program that as long as the x and y of the center of the square are not equal to the x and y of the center graphic, perform the two if then statements. The 'if then' statements check to see which direction to move the square so that it will always move toward the center.

Save your main.lua and give it a try.

Wow, that was fast! It didn't really animate, did it? Corona moved the square object so quickly, we didn't even see the movement.
Let's try a different approach. The problem is that Corona will move the square as fast as the processor can process the movement. On an older, slow smart phone, it might look okay, but on a newer one, it will move the square so fast that we don't really see it happen.

Let's use a different command that will give us the ability to move the square smoothly from its starting location to its new location in the center of the screen. If you are familiar

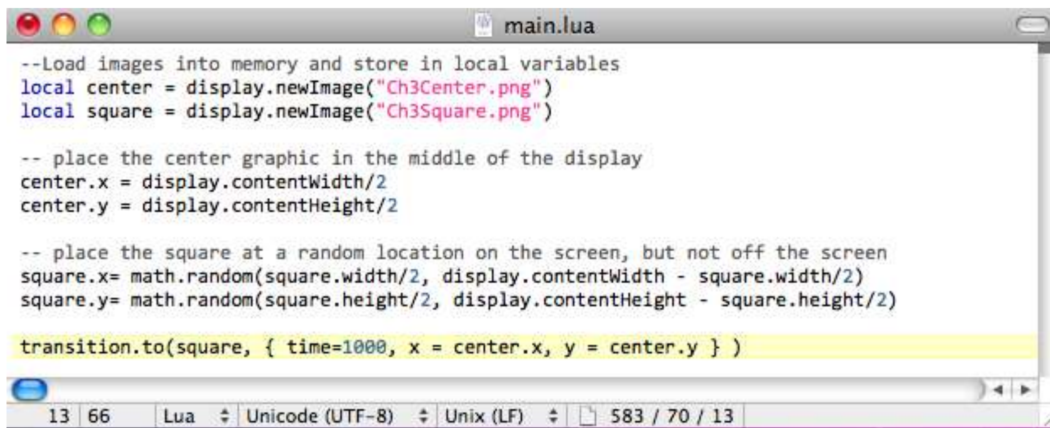
with Flash, then you have probably used tweening. In Corona, we can create similar transitions using the transition.to command:

```
transition.to( object, {array} )
```

Within the array of the transition.to parameters, we pass what we would like to change and how quickly we want that transition to occur.
My new movement code (replacing the old while loop) looks like:

```
transition.to(square, { time=1000, x = center.x, y = center.y }
)
```

In this one line of code, I have passed the square as the object. In the array, I'm passing it a time parameter that is set to 1000 milliseconds (or 1 second), and then the x and y variables.



Now You See It, Now You Don't

Being able to hide objects on the screen until they are needed is an easy way to simplify the User Interface (UI) in your apps. In Corona, the easiest way to hide an object until needed is with the alpha property. Alpha is also commonly used in game environments to cause objects to not be visible or only partially visible. At 0, an object is invisible or hidden at 1 (or 100%) an object is fully visible. You can also set the alpha at any decimal between 0 and 1 to partially fade in or out the object.

Project 3.1: Alpha Fun

For this project we are going to load three buttons: Hide, Fade, and Show. Each of these buttons will adjust the square used in the previous project by changing the alpha value in a function.

To begin with, we will need to load and place the square and each of the buttons somewhere on the display. For simplicity, I have placed them all in the center of the device.

```
--Load square png into the variable and locate it toward the
top-middle of the device.
local square = display.newImage("Ch3Square.png")
square.x = display.contentWidth/2
square.y = 50

--Load the buttons and locate them toward the bottom center of
the device.
local hideButton = display.newImage("Ch3HideButton.png")
hideButton.x = display.contentWidth/2
hideButton.y = display.contentHeight - 300
local showButton = display.newImage("Ch3ShowButton.png")
showButton.x = display.contentWidth/2
showButton.y = display.contentHeight - 200
local fadeButton = display.newImage("Ch3FadeButton.png")
fadeButton.x = display.contentWidth/2
fadeButton.y = display.contentHeight - 100
```

Then we will need to setup the function for each button. The first two, hideButton:tap and showButton:tap will be just set the alpha of the square object to either 0 or 1 (hide or show).
The third function, fade, will need to use the transition.to command to fade the square over a 3 second time span.

```
function hideButton:tap(event)
     square.alpha = 0
end

function showButton:tap(event)
     square.alpha = 1
end

function fadeButton:tap(event)
     transition.to(square, {time=3000, alpha=0})
end
```

And finally, after our functions, we will need the three event listeners, one for each button:

```
hideButton:addEventListener("tap", hideButton)
showButton:addEventListener("tap", showButton)
fadeButton:addEventListener("tap", fadeButton)
```

Save your main.lua file and give it a try.

I know. The fade button only fades out. It doesn't fade in. Let's adjust the function fadeButton:tap so that it will fade the button in if it is currently faded out.
This can easily be accomplished by adding an if then statement to check for the current alpha state of the square:

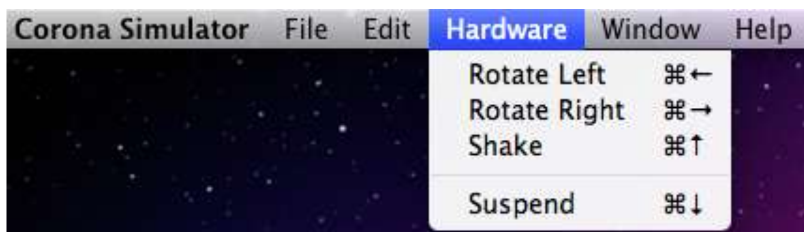
```
function fadeButton:tap(event)
      if square.alpha == 1 then
       transition.to(square, {time=3000, alpha=0})
      else transition.to(square, {time=3000, alpha=1})
     end
end
```

And there we have it! You can now fade in or out an object. Remember, the alpha property is available for all objects, so anything can be hidden until it is needed.

Orientation change

Device orientation is a very important issue to Apple and Android. They (the users and the reviewers/approval department at Apple) expect your app to work correctly in left landscape, right landscape, and portrait view if it is a phone. On tablets, your app should work in any orientation, including upside down.
That isn't to say that you can't limit your app to just landscape or portrait, but there should be a reason why it only works in that orientation. Most games have little problem being approved with having just one orientation.

There are two issues with orientation change. The first is detecting that the orientation of the device has changed. The second is changing the layout of your application for the new orientation.
As far as simulating the orientation change, it can easily be accomplished with the Corona Simulator. Through the Hardware Menu, select Rotate Left or Rotate Right.



Please recognize that this is for general rotation. Seldom will this be sufficient to handle all rotation needs of your app. You will usually need to code in the screen size and where you want the object to be located in the new orientation.

If possible your app should support every orientation based upon the device it is going to be deployed. Phones should never support upside-down orientation as it may cause confusion should the phone need to be answered.

Supported Orientations Based Upon Device

	Phones	**Tablets**
Portrait	X	X
Portrait – Upside-down		X
Landscape-Left	X	X
Landscape-Right	X	X

Project 3.2: A New Orientation

For this project, we are going to create two text objects, one that says Portrait, the other Landscape, that only show in the appropriate orientation. We will remove the incorrect text object by setting its alpha to 0, and change the appropriate object's alpha to 1.
Create a new folder for the project, and save a main.lua file to the folder.

Note: At the time of this writing, orientation events are not generated for Android devices.

To get started with our code, create your two text objects, set their color to white and the alpha of landscape to 0 (so that it doesn't show on the screen yet) and portrait's alpha to 1 (yes, we are going to assume that the app starts in the portrait orientation).

```
local portrait = display.newText("Portrait",
display.contentWidth/2, display.contentHeight/2,
native.systemFont, 24)
local landscape = display.newText("Landscape",
display.contentWidth/2, display.contentHeight/2,
native.systemFont, 24)
portrait:setTextColor(255,255,255)
portrait.alpha = 1
landscape:setTextColor(255,255,255)
landscape.alpha = 0
```

If you run the app right now, just 'Portrait' will show. Next, we need to create a function that will fire on an orientation change event and pass the new orientation to the program:

```
local function onOrientationChange (event)
    if (event.type =='landscapeRight' or event.type ==
'landscapeLeft' ) then
         portrait.alpha = 0
         landscape.alpha = 1
    else
          portrait.alpha = 1
```

```
        landscape.alpha = 0
    end
end
```

In this case, event is a parameter passed into the function from the event listener (which we will add in a few moments). event.type for an orientation change can pass:

- “portrait"
- "landscapeLeft"
- "portraitUpsideDown"
- "landscapeRight"
- "faceUp"
- "faceDown"

In our function, we are checking for the “landscapeLeft” and “landscapeRight”, which simplifies our if then statement considerably. Of course, it wouldn’t be much work to change the if then statement so that it looks for each of the possible orientation changes.

Finally, we need to add the event listener for the orientation change:

```
Runtime:addEventListener( "orientation", onOrientationChange )
```

If you save and run the app right now, you will see that it does work, but maybe not the way that we would like.

To handle the rotation of the text object, we will need to add a few more lines of code to our function. There is a second property to the event object that will help us handle the rotation of any object. event.delta returns the difference between the start and finish angles of the device, allowing the rotation to be handled very easily:

```
local newAngle = landscape.rotation - event.delta
transition.to( landscape, {time= 150, rotation = newAngle})
```

Of course, rotation can be used as a property of any object at any time; we are introducing it here to make adjusting for device orientation easier. There is a rotate method and a

rotation property. The rotation property is used to get or set the rotation of the object. The rotate method adds the specified degrees to the current rotation of the object.

The final code onOrientationChange function should now look like:

```
local function onOrientationChange (event)
    if (event.type =='landscapeRight' or event.type ==
'landscapeLeft' ) then
     local newAngle = landscape.rotation – event.delta
     transition.to( landscape, {time= 150, rotation = newAngle})
     portrait.alpha = 0
     landscape.alpha = 1
    else
     local newAngle = portrait.rotation – event.delta
     transition.to( portrait, {time= 150, rotation = newAngle})
     portrait.alpha = 1
     landscape.alpha = 0
   end
end
```

Save and run. Hmm, not quite what we want yet, is it? The problem is that since we are looking at two objects, the change in the event.delta is only updating for the last rotation. There are several ways this could be corrected. We can keep track of how many orientation changes have occurred and pass that to our rotation. We could use just one text object, changing the text on each rotation. Or we could rotate both objects each time, so that they are both always in sync:

```
local function onOrientationChange (event)
    if (event.type =="landscapeRight" or event.type ==
"landscapeLeft") then
          local newAngle = landscape.rotation - event.delta
          transition.to( landscape, {time= 150, rotation =
newAngle})
          transition.to( portrait, {rotation = newAngle})
          portrait.alpha = 0
         landscape.alpha = 1
    else
          local newAngle = portrait.rotation - event.delta
          transition.to( portrait, {time= 150, rotation =
newAngle})
          transition.to( landscape, {rotation = newAngle})
          portrait.alpha = 1
         landscape.alpha = 0
     end
end
```

And there we have a functional app that will detect orientation change!

Here is the full program just in case you missed something:

```
--Declare two text objects, set one to white and make one not
visible
local portrait = display.newText("Portrait",
display.contentWidth/2, display.contentHeight/2,
native.systemFont, 24)
local landscape = display.newText("Landscape",
display.contentWidth/2, display.contentHeight/2,
native.systemFont, 24)
portrait:setTextColor(255,255,255)
portrait.alpha = 1
landscape:setTextColor(255,255,255)
landscape.alpha = 0

local function onOrientationChange (event)
    if (event.type =="landscapeRight" or event.type ==
"landscapeLeft") then
          local newAngle = landscape.rotation - event.delta
          transition.to( landscape, {time= 150, rotation =
newAngle})
          transition.to( portrait, {rotation = newAngle})
          portrait.alpha = 0
         landscape.alpha = 1
    else
          local newAngle = portrait.rotation - event.delta
          transition.to( portrait, {time= 150, rotation =
newAngle})
          transition.to( landscape, {rotation = newAngle})
          portrait.alpha = 1
         landscape.alpha = 0
     end
end
```

```
Runtime:addEventListener( "orientation", onOrientationChange )
```

Summary

In chapter 3, we examined how to do animation with a loop and the better way of using 'transition.to'. Then we looked at using the alpha to hide or fade objects on the screen. Finally we examined how to detect for a device orientation change and using the rotation property to change the rotation of an object.

Assignments

1. Using a function, modify the project 3.1 Alpha Fun, so that the square is randomly repositioned to a new location and moves toward the center.
2. Adjust project 3.2 to use only one text object instead of two, making the appropriate changes to the function.
3. Using Assignment 3 from chapter 2, reorganize the 10 number buttons based upon device orientation. Make sure to leave room at the side for additional buttons and room at the top to display the number tapped.
4. Load 3 different buttons with different colors. Using alpha and orientation change, reorganize the buttons when the simulator's orientation is changed by setting the alpha of each button to zero, then use a transition.to to move the button to its new location, fading it in as it moves.

Chapter 4
Fill in the Blanks

In Chapter 4 we are going to examine ways to enter information into a local device. This will include:

- Using and dismissing the native keyboard
- Entering information into a textfield
- Building for and deploying to devices.

TextField

TextField is a part of the native user interface for the Apple iOS and Android devices. What does this mean? That the textfield (along with a lot of other component objects) are all a part of the individual operating systems of the different mobile devices and not a part of the OpenGL canvas that Corona uses. Apple and Android both have the ability to use a textfield and a number of other default objects such as tabs, tables, sliders, etc. Since this is a built-in part of the operating system, the Corona simulator isn't of much use to us. We have to build and port it to the device we want to test it on.

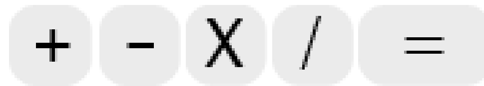
Yes, that's right; in this project you are going to get to load your first app on to an actual device! First we will walk through the programming, then tackle each operating system build separately.

The textfield is used for a single-line of text input. As it is not part of the OpenGL canvas, it does not play well with Corona's display object hierarchy. What does that mean for you as a developer? Basically that while you can change a textfield's location, it will always appear above (or in front of) all other objects on the screen.

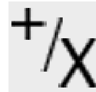
Project 4: Simple Calculator

In this first app, we are going to use textboxes to enter numbers that we would like to add, subtract, multiply, or divide. The numbers will be entered into a textfield, and the user will be able to specify what operation they would like to occur, and then see their result.

To get started create your project folder and main.lua file. We will need 5 button objects for this project: one each for add, subtract, multiply, divide, and the equals sign. I made these five buttons in Photoshop. You can either make your own or use the graphics contained in the downloadable file at http://www.BurtonsMediaGroup.com/books/. The buttons are 50 by 50 pixels, which is well within standard recommended guidelines for app development.



I have also included an icon.png file, which is required to perform builds for the Apple iOS. Icon files should be 57 x 57 pixels (114 x 114 for iPhone 4 and newer) and saved as a png file format and placed in the root folder of your app with your main.lua file. Icons for Android devices should be named: Icon-hdpi.png, Icon-mdpi.png, and Icon-ldpi.png with sizes of 72x72, 48x48 and 36x36, respectively.



First, let's load a clear background and the five buttons into our app:

```
-- load buttons and place in on the display
local bkgd = display.newImage("bkgd.png",0,0)
local width = display.contentWidth/2
local height = display.contentHeight/2 - 100
local addButton = display.newImage("add.png", width -145,
height)
local subtractButton = display.newImage("subtract.png", width -
60, height)
local multiplyButton = display.newImage("multiply.png", width
+15, height)
local divideButton = display.newImage("divide.png", width + 85,
height)
local equalButton = display.newImage("equals.png", width, height
+ 150)
```

The bkgd variable now holds a transparent image that I have set to 960 X 640 pixels – the same size as an iPhone 4 display. Since that is the largest area I expect to have to cover (assuming we don't deploy to a tablet), it will work fine for any other display. I set the bkgd's top left corner to be a 0, 0. This ensures full coverage of the screen. It must be loaded first to ensure that it is at the bottom of the stack of graphics (i.e. behind everything else). We will later use the bkgd object to tap on so that we can dismiss the keyboard. Next we load the buttons and place them toward the middle of the screen. Make sure to leave room to place a textbox above and below the row of buttons and space to display the answer below the equals sign. Notice that I used the width and height variables to calculate where each button would be placed. This allows the app to work on multiple devices, as long as I take care to ensure that everything is located within the bounds of my lowest resolution device (an iPhone 3G/3GS, at 480 x 320).

Wanting to get as much coding in before I have to deploy as possible, I am going to insert placeholder text where my textboxes will go. It is important to use your simulator for as much as you can for as long as you can to speed up app development. While it only takes a few minutes to deploy to a device (significantly longer the first time through), those

minutes quickly add up and cut into your productive coding time. To that end, I am also going to add my event listeners and functions to perform the calculations. This way I can ensure all of my program logic is working before I deploy to a device. Then, if I have problems, I know they are device related, not my programming logic.

```
 -- Textbox for first number
local firstNumber = 10
local firstNumberText = display.newText(firstNumber, width +
100, height - 75, native.systemFont, 36)

-- Textbox for second number
local secondNumber = 5
local secondNumberText = display.newText(secondNumber, width +
100, height +75, native.systemFont, 36)
local operator   -- variable to tell us which operator was
selected
local result     -- variable to hold the result
local operandSelected = "False"
local resultText  --variable to hold the result text to be
displayed.
```

In this section we have set some test numbers (firstNumber and secondNumber), and created a couple of variables to keep track of our operand and results. For added error checking, I have a variable to check to make sure that an operand has been selected so that we can be sure that a calculation can be performed when the '=' sign is pressed.

```
local function addButtonTap(event)
     operator = "+"
     operandSelected = "True"
end
local function subtractButtonTap(event)
     operator = "-"
     operandSelected = "True"
end
local function multiplyButtonTap(event)
     operator = "*"
     operandSelected = "True"
end
local function divideButtonTap(event)
     operator = "/"
     operandSelected = "True"
end
```

Next we code the functions for each button, designating which operator will be used and setting the operandSelected variable to true. After the operand is coded, we can move on to the equalButtonTap function:

```
local function equalButtonTap(event)
if operandSelected == "True" then
     if operator == "+" then
          result = tonumber(firstNumber.text) +
tonumber(secondNumber.text)
     elseif operator == "-" then
          result = tonumber(firstNumber.text) -
tonumber(secondNumber.text)
     elseif operator == "*" then
          result = tonumber(firstNumber.text) *
tonumber(secondNumber.text)
     elseif operator == "/" then
          result = tonumber(firstNumber.text) /
tonumber(secondNumber.text)
     end

     local resultText = display.newText(result, width, 370,
native.systemFont, 36)
     resultText:setTextColor(255,255,255)
     operandSelected = "False"
    else
     local warningText = display.newText("Select operation
first", width-150, 100, native.systemFont, 36)
     warningText:setTextColor(255,255,255)
    end
end
```

The equalButtonTap function first checks to ensure that an operator has been selected, if it hasn't then a warning message is displayed at the top of the screen. Next we use 'if then

elseif' to calculate the result. We use the tonumber() function around each of the variables to convert to ensure that the number that is stored is treated as a number, not a string. Finally the result is displayed under the equals sign and the operandSelected is reset to "False".

The last step to before testing the code in the simulator is to create a variable that will dismiss the keyboard after the user has entered a value (note: code for the keyboard is not needed in the simulator run, since the numbers 10 and 5 are hard coded). Finally, add the event listeners for the various buttons and a line of code for the background ('bkgd') listener that takes advantage of the closure capability of Lua to create a dynamic function to check on the keyboard focus. While we will discuss dynamic functions more in a later chapter, for now it is enough to understand that you are assigning a function to a variable name listener. If that function is called (through the action of the user of the app tapping the background ('bkgd'), the native keyboard is dismissed by setting it to nil.

```
local listener = function (event)
     native.setKeyboardFocus(nil)
end

addButton:addEventListener("tap", addButtonTap)
subtractButton:addEventListener("tap", subtractButtonTap)
multiplyButton:addEventListener("tap", multiplyButtonTap)
divideButton:addEventListener("tap", divideButtonTap)
equalButton:addEventListener("tap", equalButtonTap)
bkgd:addEventListener("tap", listener)
```

When we have checked the functionality of our code in the simulator, we can proceed to add the textfields and perform a device build.

Comment out the firstNumber and secondNumber code. Replace these hardcoded numbers with:

```
native.newTextField( left, top, width, height [, listener ] )
```

and add the commands as shown below:

```
-- Textbox for first number
local firstNumber = native.newTextField(width - 100, height -
75, 220, 36)
firstNumber.inputType= "number"

-- Textbox for second number
local secondNumber = native.newTextField(width - 100, height +
75, 220, 36)
secondNumber.inputType= "number"
```

As you may have guessed, we just told Corona to display two different text fields and have set the input keyboard for both to numeric. The keyboard input type can be set to:

- “default” – a standard keyboard supporting general text, numbers, and punctuation.
- “number” – a numeric keypad.
- “phone” – a keypad for entering phone numbers.
- “url” – keyboard for entering website URLs.
- “email” – a keyboard for entering email addresses.

After adding these lines of code and saving, you should get a warning message in the terminal informing you that the native text field is not supported in the simulator. One last change before we actually build is to add .text to each of the textfields for calulations so that the number is returned. I have bolded the changes in the final code below:

Final Code for Calculator:

```
-- Project 4 Calculator

-- load buttons and place in on the display
local bkgd = display.newImage("bkgd.png",0,0)
local width = display.contentWidth/2
local height = display.contentHeight/2 - 100
local addButton = display.newImage("add.png", width -145,
height)
local subtractButton = display.newImage("subtract.png", width -
60, height)
local multiplyButton = display.newImage("multiply.png", width
+15, height)
local divideButton = display.newImage("divide.png", width + 85,
height)
local equalButton = display.newImage("equals.png", width, height
+ 150)

-- Textbox for first number
--local firstNumber = 10
--local firstNumber = display.newText(firstNumber, width + 100,
height - 75, native.systemFont, 36)
local firstNumber = native.newTextField(width - 100, height -
75, 220, 36)
firstNumber.inputType="number"

-- Textbox for second number
--local secondNumber = 5
--local secondNumber = display.newText(secondNumber, width +
100, height +75, native.systemFont, 36)
local secondNumber = native.newTextField(width - 100, height +
75, 220, 36)
secondNumber.inputType="number"
```

```
local operator   -- variable to tell us which operator was
selected
local result     -- variable to hold the result
local operandSelected = "False"
local resultText
local warningText

local function addButtonTap(event)
     operator = "+"
     operandSelected = "True"
end

local function subtractButtonTap(event)
     operator = "-"
     operandSelected = "True"
end

local function multiplyButtonTap(event)
     operator = "*"
     operandSelected = "True"
end

local function divideButtonTap(event)
     operator = "/"
     operandSelected = "True"
end

local function equalButtonTap(event)

   if operandSelected == "True" then
     if operator == "+" then
          result = tonumber(firstNumber.text) +
tonumber(secondNumber.text)
     elseif operator == "-" then
          result = tonumber(firstNumber.text) -
tonumber(secondNumber.text)
     elseif operator == "*" then
          result = tonumber(firstNumber.text) *
tonumber(secondNumber.text)
     elseif operator == "/" then
          result = tonumber(firstNumber.text) /
tonumber(secondNumber.text)
     end

     local resultText = display.newText(result, width, 370,
native.systemFont, 36)
     resultText:setTextColor(255,255,255)
```

```
     operandSelected = "False"
  else
     local warningText = display.newText("Select operation 
first", width-150, 100, native.systemFont, 36)
     warningText:setTextColor(255,255,255)
  end
end

local listener = function (event)
     native.setKeyboardFocus(nil)
end

addButton:addEventListener("tap", addButtonTap)
subtractButton:addEventListener("tap", subtractButtonTap)
multiplyButton:addEventListener("tap", multiplyButtonTap)
divideButton:addEventListener("tap", divideButtonTap)
equalButton:addEventListener("tap", equalButtonTap)
bkgd:addEventListener("tap", listener)
```

Device Builds

First, as I have mentioned before, you must have a Macintosh computer to build and deploy for the Apple iOS. You can build for Android devices using either a Macintosh or a Windows PC.

Apple iOS

Apple regularly updates their process for building to a device. For the latest build updates for Corona to an Apple device, check http://developer.anscamobile.com/content/building-devices-iphoneipad

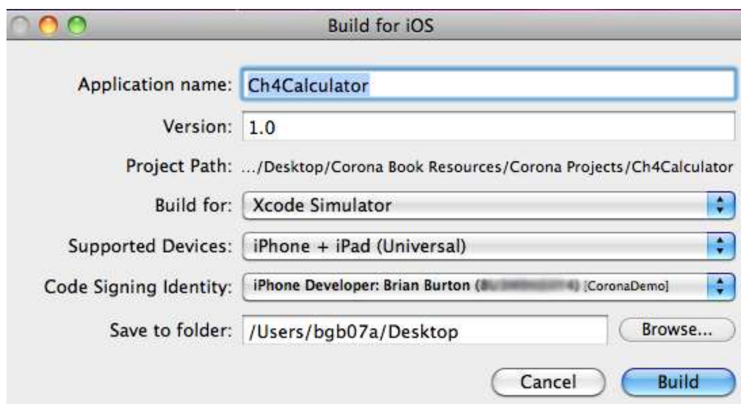
Before we begin deploying to your Apple iOS, I want to remind you that you must be a current Apple Developer with a Standard (most common), University (most common for students), or Enterprise Developers account. You will need your code signing identity/provisioning certificate already configured through the Apple Developers website to be able to build for the simulator or a device.

When building for an iOS device, you have the option of building to the iOS simulator (the simulator that comes with Xcode) or for a device. Building for the simulator provides you one more opportunity to get the bugs worked out before going through the time consuming process of deploying to a physical device. When I say time consuming, this is in comparison to clicking build and having it show in the simulator. It takes a couple of minutes each time you deploy to a physical device, and those minutes add up.

For building, we will walk through both processes, for the iOS simulator and then to the actual iPhone.

First, with your Corona Simulator selected, Click on File > Build > iOS… (or Command-B) to open the Build for iOS window.

iOS Simulator Build



Verify your application name and version number, then change the 'Build for:' dropdown to Xcode Simulator instead of device. Change your supported Devices to iPhone only for this walk-through. Finally, select your Code Signing Identity. When you click on Build, the iOS Simulator will launch. Under the hardware menu, you can change the device to be simulated and the version of OS to simulate.

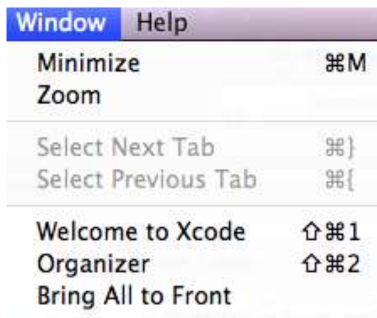
Apple iOS Device Build

While performing simulator builds can help us quickly determine problems with our app, there is something particularly rewarding about seeing your app (even a simple app such as our calculator) on the actual device.

When building for the iOS device, there are a few more steps involved. To begin with, instead of Build for Xcode Simulator, you will need to select Device as shown:

On supported devices, I recommend selecting the device you plan to deploy to instead of a universal build. Universal builds are great when you are ready to go to market, but at this point in our trouble shooting, building for just an iPhone or iPad is easier.

After you Build, open Xcode. Under the Window menu item select Organizer.



This will open the device organizer. If you have already configured your devices and provisions, you will be able to quickly add the app to your device. Under the device that you are using to test your app, select Applications. Then drag your Ch4Calculator build on the Organizer. This will install the app to your device. How long it takes will depend upon the size of the app, but our calculator should deploy fairly quickly. When you see the Ch4Calculator 1.0 listed, it will be deployed to your test device. You can then try it on your device.

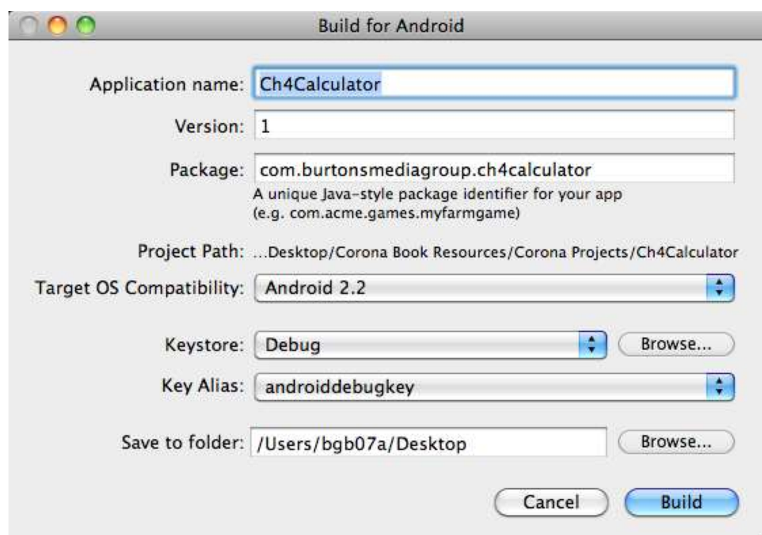
Android OS Device Build

To build for an Android device, the device you are deploying to must have Android 2.2 or newer installed and have an ArmV7 processor. There are multiple ways to build for an Android device. I am going to cover the three most popular: command line, web server, and dropbox. The first step, no matter which method you use to deploy to the device, is to build the package.

In your Corona Simulator, select File > Build > Android which will open the Build for Android dialog box.

Make sure you are happy with the application name and set the version number. For 'Package,' use a java-style package identifier, which is a reverse URL with the app name. If you don't have a URL for your apps, you can use anything here, but be aware that you MUST have a support website when you go to start marketing your apps. So add it to your To Do list: make a website for all of your awesome apps!

My package identifier is: com.burtonsmediagroup.ch4calculator



Your target OS should already be set to Android 2.2. Make sure your Key Alias is set to androiddebugkey and click on Build.
This will create Ch4Calculator.apk which can be deployed to your test device. As I mentioned earlier, there are a number of ways to deploy the app to your test device.

Method #1:
If you have the Android SDK installed, you can use the command line:
```
adb install Ch4Calculator.apk
```
to your USB connected device.

Method #2:
Upload your app to your webserver (see, I told you that you would need one!) and point your Android device's web browser to the file's URL. This will allow you to download and install the file to your phone.

Method #3:
This is by far the easiest method in my opinion. Get a DropBox account from http://www.dropbox.com and install it to your development computer and your android device. Once you have it configured, copy the .apk file to your dropbox on your development system. Then on your android device browse to the folder you placed your .apk file, click on it to download and install.

I should note that if you develop on multiple systems like I do, having a dropbox account makes it very easy to transfer development files back and forth between systems.

While we could add a lot more code for error checking and to ensure that the user has entered a non-zero value, we have accomplished our goal of creating a simple calculator and building it for a device.

Summary

Once again we have packed a lot of information into a few short pages. In this chapter we discussed the textField native object, and setting it to different keyboards. We also examined how to dismiss a keyboard by tapping on the background using a transparent background image.
We then covered the different methods of deploying to an iOS and Android device.

Assignments

1. Add additional mathematical functions to the calculator such as square root, exponential, tangent, sine and cosine. The calculator should also perform basic error checking such as divide by zero.
2. Modify the Simple Calculator app so that it will work in either landscape or portrait views.
3. Revise the calculator app by replacing it with number keys instead of textboxes for number entry.
4. Create an app that allows you to enter email addresses using the appropriate keyboard.
5. Deploy an app to an iOS device and Android device. What is different about how the app looks and operates between the two devices?

Chapter 5
All Things Graphic

One of the things that many people find appealing about Corona is how easy it is to create and load graphics into the mobile environment. In this chapter we are going to:

- Create vector based graphics
- Load bitmap graphics
- An introduction to sprite sheets
- Review associated graphic properties

We all know it is the driving force of why smart phones are popular; the ability to create interactive graphics. In this chapter, we are going to look at how to draw basic graphic shapes with vector graphics and how to work with bitmap graphics created in other software such as photoshop or gimp. We will also examine how to use sprite sheets in Corona.

Vector Graphics

A vector graphic is a geometrical primitive (such as a line, curve, circle, or rectangle) that is based upon a mathematical equation. Vector graphics are the smallest files and the fastest images to display (as far as drawing to the screen) and are able to be resized or scaled infinitely since the shape is based upon a math equation instead of a bitmap image comprised of pixels.

There are three basic vector:

- display.newCircle(xCenter, yCenter, radius) – creates a circle at xCenter, yCenter with the given radius.
- display.newLine(x1, y1, x2, y2) – draws a line from the first point to the second point. You can append line segments with the :append method.
- display.newRect(left, top, width, height) – creates a rectangle starting at the location given for the top, left corner. Width and height parameters are absolute pixel lengths (i.e. they set the height and width off of the top, left corner location)
- display.RoundedRect(left, top, width, height, cornerRadius) – like the newRect except with rounded corners. CornerRadius sets the quarter radius of each corner.

Vector-based objects, with the special exception of newline, all have a default reference point at their respective center. They all have the following properties or methods that can be set:

- object.strokeWidth - Sets the width of the line in pixels

- object:append() - Appends one or more line segments to an existing display.newLine object.
- object:setColor() - Sets the color of a line object based upon r, g, b (and optionally alpha) values between 0 and 255
- object:setFillColor() - Sets the fill color for vector objects based upon r, g, b (and optionally alpha) between 0 and 255

Project 5: Vector Shapes

For this first graphics project, we are going to create each of the vector shapes, then use the methods and objects available to manipulate them in the display environment. Create a folder and main.lua file to start.

We will start by finding the center of the display and storing it in the variables w and h. Then we will create a star shape using a line segment and appending the additional lines to the initial segment. After we draw the star, we will set the the stroke color to white and the stroke to a 3 pixel width.

main.lua

```
-- Store the center of display for later use
local w = display.contentWidth/2
local h = display.contentHeight/2

-- Star shape: need initial segment to start
-- newline accepts the start x, y and end x, y of line
local star = display.newLine( 0,-110, 27,-35 )

-- further segments can be added later
star:append( 105,-35, 43,16, 65,90, 0,45, -65,90, -43,15, -105,-
35, -27,-35, 0,-110 )
star:setColor( 255, 255, 255, 255 )
star.strokeWidth = 3
```

As you might have noticed, a portion of the star is off the screen, but we will bring it into view shortly. Next, we will add a rectangle and circle to the display.

```
local rectangle = display.newRect( 100, 100, 50, 50)
rectangle.strokeWidth = 5
rectangle:setFillColor( 255, 0, 0)
rectangle:setStrokeColor(0, 0, 255)

local circle = display.newCircle(display.contentWidth/2,
display.contentHeight/2, 15)
circle.strokeWidth = 2
circle:setFillColor(0, 255, 0)
circle:setStrokeColor(255,255,255)
```

Our new rectangle object's - **newRect(left, top, width, height)** – the left, top corner of the rectangle is at 100 down and 100 pixels over from the top left corner of the device display. The rectangle has a width of 50 pixels and a height of 50 pixels (so it is a square). We set the fill color of the rectangle to red, its line stroke (outline) color to blue, and the pixel width of the rectangle outline to 5 pixels.

The circle – newCircle(x Center, y Center, radius) – is located in the center of the screen with a radius of 15 pixels, a stroke width of 2, a fill color of green, and a stroke color of white.

Now that we have these three objects added to our display, we will move them using the transition.to command to the center of the screen.

```
transition.to(star, {x=w, y=h, time=1500})
transition.to(rectangle, {x=w, y=h, time = 1500})
```

Notice two things: first, the objects stack in the order that they were loaded: with the star in the background, the rectangle in the middle, and the circle on top.
Second, the star is not 'centered' like the rectangle and circle. This is because the star's location is based upon the first line that was drawn, not the center of the collection of lines. For our star, this has the effect of everything being lined up for the object's x parameter, but the y parameter is off by 110 pixels.
To correct this problem, we use the .yReference parameter. Changing yReference changes the reference point for the y of the object so that all movement and rotation are now based upon the new value instead of the original y value for the first line segment. Add

```
star.yReference = 110
```

before the transition.to commands and run your app to see the difference. As you would expect, there is also an xReference parameter that can be set should the need arise.

Let's make one more change before we move on. We used the rotation parameter previously when we were working on device orientation. Add a rotation of 360 degrees to the transition.to commands so that your final code looks like:

```
-- Vector graphics example
local w = display.contentWidth/2
local h = display.contentHeight/2

-- need initial segment to start
local star = display.newLine( 0,-110, 27,-35 )
star:append( 105,-35, 43,16, 65,90, 0,45, -65,90, -43,15, -105,-
35, -27,-35, 0,-110 )
star:setColor( 255, 255, 255, 255 )
star.strokeWidth = 3
star.yReference = 110
```

```
local rectangle = display.newRect( 100, 100, 50, 50)
rectangle.strokeWidth = 5
rectangle:setFillColor( 255, 0, 0)
rectangle:setStrokeColor(0, 0, 255)

local circle = display.newCircle(w, h, 15)
circle.strokeWidth = 2
circle:setFillColor(0, 255, 0)
circle:setStrokeColor(255,255,255)

transition.to(star, {x=w, y=h, time=1500, rotation=360})
transition.to(rectangle, {x=w, y=h, time = 1500, rotation =
360})
```

While vector graphics are very fast (processor-wise) they are time consuming to code and somewhat limited in features and complexity. Let's be honest with ourselves, can you see yourself creating a complex landscape or background with vector graphics? It might have worked for Lunar Lander back in the late 1970s but today's smartphone users expect a little more! Fortunately Corona has taken care of this issue with bitmaps!

Bitmap Graphics

A bitmap graphic is created by using a series of colored pixels to form complex (or simple) images. They are stored in external files in various formats, the most common of which are jpeg, gif, and png. PNG is the recommended bitmap format for maximum compatibility across multiple platforms.

We have been using display.newImage() for a couple of chapters now to load our bitmap graphic content into our app. There are a couple of considerations from the Corona website to keep in mind on your graphics:

- Make sure you use "Save for Web" when exporting your images. This will ensure that the image does not contain an embedded ICC profile and is an appropriate file size for a mobile device.
- To help conserve memory (always a problem when you start working on image intensive apps!) make sure that your image is between 72 dots per inch (DPI) and 170. 72 is the default for Photoshop when you start a new image.
- There may be gamma and color differences between the system you develop the graphics on and the devices you export to. Make sure your art person has calibrated their display to your export device, or that great yellow texture might not be as appealing on the device.
- Gray scale images are not currently supported. Make sure your images are RGB.
- Indexed PNG images are not supported by Corona.

- Maximum image resolution supported is 2048 x 2048. Older devices will have a lower maximum resolution (thankfully, they are becoming more rare).

The full parameter list for display.newImage() is:

```
object = display.newImage([parentGroup,] filename [,
baseDirectory] [, left, top] [,isFullResolution])
```

We will discuss parentGroup and baseDirectory in chapter 7. You should already be familiar with left and top (sets the images left and top corner). That leaves us with **isFullResolution**.

The **isFullResolution** is a Boolean parameter that overrides autoscaling (I will discuss why this is a bad idea in chapter 6) and forces the image to be shown at its full resolution. By default, this parameter is false.

Resolution

A good choice, if you plan to work with multiple resolutions and devices, is to use display.newImageRect(). The display.newImageRect() command substitutes higher-resolution assets (i.e. bitmap images) on higher resolution devices. This will be fully discussed in chapter 6.

Icons

While we are on the topic of resolutions, let's return to the discussion of icons that was begun in chapter 4. This isn't an issue if you are making your app for just one platform, but if you are leveraging your resources so that you can deploy to multiple platforms (that's what first attracted me to Corona), then you will need icons for all the required sizes by the various vendors. It is recommended that you begin all of your graphics for the largest size then scale them down as appropriate. At this time there are 6 different icon sizes that you need to be concerned with:

Android:

Icon-hdpi.png	72x72px
Icon-mdpi.png	48x48px
Icon-ldpi.png	36x36px

Apple:

Icon.png (iPhone 3G &3GS)	57x57px
Icon@x2.png (iPhone 4)	114x114px
Icon-72.png (iPad)	72x72px

Oh, and don't forget you will need a 512x512px for the iTunes store.

Scaling

Once your image is loaded, you have three ways of adjusting the scale of an object: yScale and xScale, the scale method, and using the scale method with xScale and yScale. You can use the object property xScale and yScale, which scales the object based upon the objects reference point. For most objects, this is the center of the object. Use the scale(sx, sy) method to set the xScale and yScale properties. Each time you modify the scale using the scale method, the object is multiplied times the value xScale and yScale. If xScale and yScale have not been set, they default to the value of 1.

Example:

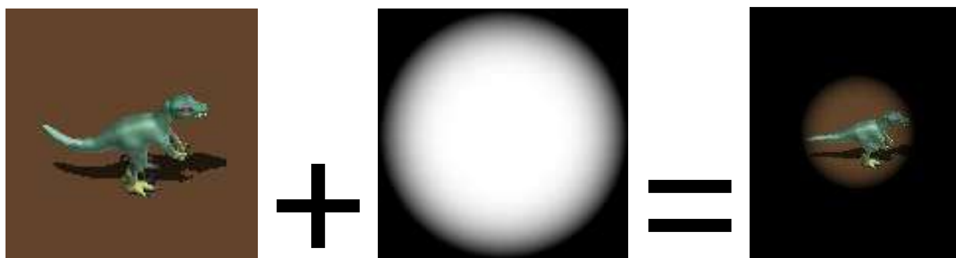
```
myImage.xscale = .5
myImage.yscale = .5

myImage:scale(.5, .5)
```

This short code would result in an image that was displayed at ¼ its original size after the code was completed. Why, you might ask? When you call the scale method, it will multiply the set scale by any previous scale that has been set. So we get the result of: .5 x .5 = .25. This is an important item to remember if you use the scaling methods.

Masking

Masking allows you to hide a portion of your screen by placing one graphic in front of another. Masking is a very powerful tool and can be used to create spectacular effects in your apps.

A mask is always associated with another object, whether it is another graphic, text, or a display group (discussed in chapter 7). Masks can also be nested.



To create your own mask, you will need to create a bitmap image that will cover a portion of the object to be masked. You can think of it like a ballroom mask. If you desire to hide a portion of your face, you need to decide what portions will be visible and what will be hidden. When you are creating your mask image, dark areas will cover or hide the covered object and white areas will be clear or not hidden. Load the mask using:

```
local mask = graphics.newMask(filename)
```

To apply the mask to an image:

```
image:setMask (mask)
```

graphics.newMask converts the image to gray scale with the black values acting as masks, and the white values becoming transparent. Anything outside the mask is filled with black pixels (thus masking the rest of the screen). A few notes on masking:

- The mask image width and height must be a multiple of 4.
- The mask image must have a black border around the mast that is at least 3 pixels.

Masking does not impact the touch and tap events of an image. In other words, if you mask something, touch and tap events can still occur even if the object is hidden.

To set a mask to an object, you use the setMask() method: object:setMask(mask object)

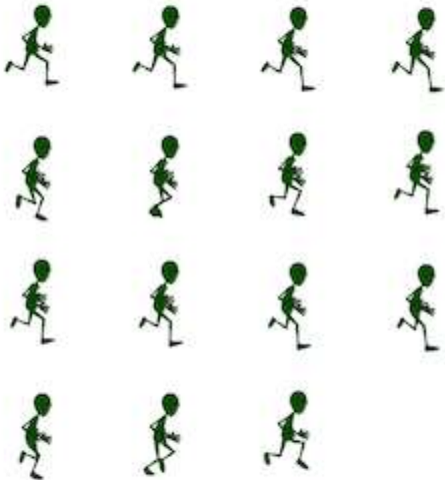
You can also rotate, scale, and set the x and y of the mask with the appropriate parameter:

object.maskRotation
object.maskScaleX
object.maskScaleY
object.maskX
object.maskY

Sprite Sheets

Sprite sheets are commonly used for games and animation. They are 2D images saved as multiple frames in a single png image file. This allows for a more effective and efficient use of memory.

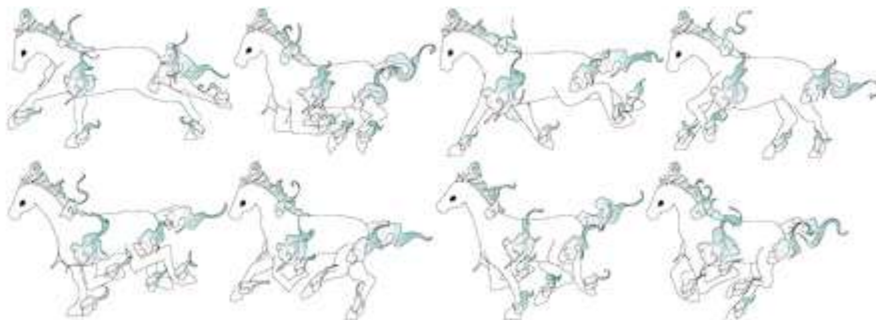
Corona provides support for two types of sprite sheets: uniform frames and non-uniform frames. Uniform frames are 2D images that are all the same size throughout the sprite sheet. The image below is an example of a uniform frame sprite sheet provided in the Jungle Scene sample project that ships with Corona.



Jungle Scene Sprite Sheet - demonstrating uniform frames

As you can see, each image is uniform in size and positioning, providing an animation sequence of the character running.

The second type, non-uniform frame sprite sheets, contain multiple images that are of varying height and width. A non-uniform frame sprite sheet stores the location and size of each frame in an external data file so that Corona is able to properly load each of the frames. Below is an example of a non-uniform sprite sheet from the HorseAnimation project that ships with Corona.



Horse Animation Sprite Sheet - demonstrating non-uniform frame sizes

To assist with your sprite sheet creation, there are many great tools available such as Spriteloq, Zwoptex, and TexturePacker. More information is provided on each of these programs in chapter 17.

To use the sprite API in a Corona project, you must place the command

```
require “sprite”
```

at the beginning of your main.lua file.

Sprites are incredibly useful and powerful for game creation. We will examine how to create a game project in chapter 16 that make use of Sprite Sheets. For now, we will combine what we have learned about masking and sprite sheets to create a simple project.

Project 5.1: Uniform Sprites

For this project we are going to create a simple app that displays a sprite. The sprite is stored in greenDinoSheet.png with the parameters for each sprite stored in greenDinoSheet.lua. The green dino sprites include a walk, a look, roar, and run sequence. The green dino sprites are from Reiner's Tile sets (http://www.reinerstilesets.de).

Uniform space and size sprite sheet

Adding a sprite to your app is fairly simple once you have the sprites.

```
local sprite = require("sprite")
```

After setting sprite to equal sprite library, you will need to load the sprite sheet into memory. There are two types of sprite sheets as has previously been mentioned: uniformly spaced sprites and non-uniformly spaced sprites. Our greenDino has a uniform size of 128x128 pixels, so we can use newSpriteSheet to load the sheet in to memory, passing it the file name of the sprite sheet and the frame width and height. We will look at how to load non-uniform (the more common) sprites in Project 5.2.

```
local sheet1 = sprite.newSpriteSheet("greenDinoSheet.png", 128,
128)
```

Next, we create a sprite set. A sprite set is a collection of frames that are related. Often sprite sheets will contain multiple groups of animations and/or characters. newSpriteSet requires the name of the sprite sheet, start frame of the animation, and the number of frames in the animation. In the example below, my sprite sheet is stored in sheet1, the start frame is 1, and the frame count is 8.

```
local spriteSet1 = sprite.newSpriteSet(sheet1, 1, 8)
```

Next, we will add a named sequence. A named sequence allows us to specify "walk," "run," "jump," "attack," "die," or any other name to our sequence, which is much easier to use in game design than trying to remember your start and end frame. In our example below, we are adding from the spriteSet1, and giving it the name "walk." It will start with frame 1 and run through 8 frames. The 1000 refers to how long the app will take to play the sequence of 8 frames in milliseconds.

The last parameter we have is the loop parameter. A value of 0 (the default) means the sequence will loop indefinitely. A value greater than 0 will result in the sequence playing the specified number of times. A -1 means the sequence will "bounce" back and forth once, playing the frames up to the final frame, then playing them in reverse order (1, 2, 3, 2, 1) and stop. A -2 value means the sequence will bounce forever.

```
sprite.add(spriteSet1, "walk", 1, 8, 1000, 0)
local instance1 = sprite.newSprite(spriteSet1)
instance1.x = display.contentWidth/2
instance1.y = display.contentHeight/2
instance1:prepare("walk")
instance1:play()
```

After adding the sprite, we create an instance of the sprite as a display object. Once the instance is created, we set its x and y location, execute a prepare method which stops any currently playing animation sequence and moves to the first frame of the specified sequence. Finally we execute the play method.

The brown area rips because it is a background area. I have left it in for you to see the impact of not setting the background of a sprite to transparent or not having the same background color throughout your app (both of which would fix the apparent 'ripping' of the brown background).

Project 5.2: Non-uniform Sprites

Non-uniform sprites are actually very easy to work with IF you use a program like Spriteloq, TexturePacker, or Zwoptex to help you manage creating the required data file. The data file is a lua format file that describes the sizes and positions of each sprite stored in the associated sprite sheet image file:

Sample sprite sheet data file:

```
function getSpriteSheetData()
    local sheet = {
        frames = {
            {
                name = "roaring w0000.bmp",
```

```
                spriteColorRect = { x = 25, y = 44, width = 92,
height = 52 },
                textureRect = { x = 0, y = 204, width = 92,
height = 52 },
                spriteSourceSize = { width = 128, height = 128
},
                spriteTrimmed = true,
                textureRotated = false
            },
```

Non-uniform space and size sprite sheet created with TexturePacker

For this example, I will use the same main.lua file from Project 5.1. First, let's change the

```
sprite.add(spriteSet1, "walk", 1, 8, 1000, 0)
```

to

```
sprite.add(spriteSet1, "walk", 1, 8, 1000, -1)
```

which gives the appearance of the green dino walking forward, then walking backwards. Next, we will add a mask around the green dino.

```
local mask = graphics.newMask("circlemask.png")
instance1:setMask(mask)
```

As you can see, the mask is a little too big. Let's scale it down by 50%.

```
instance1.maskScaleX = .5
instance1.maskScaleY= .5
```

Now, we will add the second dino. First we will setup a variable to hold the data sheet information which is stored in redDinoSheet.lua, so it must be loaded with the require command. Then load the sprite sizes and positions into the variable spriteData using the getSpriteSheetData function that is stored in redDinoSheet.lua.

```
local sheetData = require("redDinoSheet")
local spriteData = sheetData.getSpriteSheetData()
local sheet2 = sprite.newSpriteSheetFromData("redDinoSheet.png",
spriteData )
```

Sheet2 is set equal to the sprite png file and associated with the data stored in spriteData. Finally, the remaining code works just like the green dino sprite.

```
local spriteSet2 = sprite.newSpriteSet(sheet2, 1, 9)
sprite.add(spriteSet2, "redRoar", 1, 9, 1000, 0)
local instance2 = sprite.newSprite(spriteSet2)
instance2.x = display.contentWidth/2 + 128
instance2.y = display.contentHeight/2
instance2:prepare("redRoar")
instance2:play()
```

Summary

This chapter included a number of essential elements for creating and using graphics in a Corona project. At this point you should feel comfortable creating a vector based graphic, importing sprites, using a mask, scaling and handling multiple resolutions. In our next chapter we will jump into the world of handling the user interface.

Assignments

1) Create your own stacked set of vector based graphics. Using the new line, create a pentagon and an octagon shape.

2) Using vector shapes, simulate various special effects such as an arrow, laser or bubbles.

3) Using the sprite sheets of red and green dino, create a short dramatization. Included in the green dino sheet are animation sequences for look, roar, run, and walk. The red dino also has roar and walk sequences.

4) Create an app using the green dino that walks to a tap location on the screen.

5) Create your own sprite sheet and build an app to show off your creation.

6) Create a mask that says “Corona Rocks” and place it over a multicolor image.

Chapter 6
User Interface

In this chapter we are going to look at ways to improve your User Interface and how to develop for multiple platforms more quickly and easily. This will include:

- Using the build.settings file
- Using config.lua to handle runtime configuration
- UI.lua to cut down on repetitive coding
- Adding sound to your apps

We have all worked with software that was difficult to use. The User Interface (UI) can make or break any app. If your app is a great idea but the interface doesn't allow the user to use it the way that you intended (or the way they want to use it), it will get horrible reviews and not reach the sales you would like. In this chapter we will examine how to simplify building your app for multiple platforms, using the UI.

Resources

Remember to place all of your files including graphics and sound files in the same folder. You can use subfolders to help keep your files organized, but keep all of your lua files in the root of your directory..
I recommend using the Corona Project Manager (discussed further in Chapter 17: Resources) to create and manage your projects. CPM makes projects that use a lot of outside resources (such as graphics and audio) and the additional Lua files that are discussed in this chapter much more manageable. While it isn't free, it is well worth the expense in my humble opinion.

build.settings

Corona allows you control over the build of your app through the build.settings file. build.settings uses Lua syntax to specify the default settings for your app. The build.settings file is used to set the application orientation options and auto-rotation behavior. It may also contain platform-specific parameters. The build.settings file should be created in your application folder with your main.lua file.

Sample build.settings:
```
settings =
{
    orientation =
    {
        default = "portrait",
        supported =
        {
```

```
            "portrait", "portraitUpsideDown", "landscapeRight",
"landscapeLeft"
        }
    },
}
```

This build.settings file configures the default orientation to portrait and also supports auto-rotation to all four orientations. This only impacts iOS devices. Android devices will automatically open to the orientation of the device unless only one orientation is specified. Android devices also only currently support the orientations of landscapeRight and portrait.

The build.settings is capable of a few other advanced configuration settings which I will discuss at a later time.

config.lua

Dynamic Content Scaling
You may have noticed that not all mobile devices have the same resolution. The screen estate for iPhone and iPod is 320 x 480 (see code below where I set that as the default for all devices for this app). With config.lua, you are able to allow your app to do dynamic content scaling so that your app looks and runs great on any device, even those with a higher screen resolution. The config.lua file should be included in your project folder. Note: If you are building for a single device type, it is unnecessary to have dynamic content scaling, so you can skip having a config.lua file.

To use dynamic content scaling, create a config.lua file in your project folder with your editor. You will set the width and height in pixels of your original target device, and then set your auto-scaling. Auto-scaling has four predefined settings:

- “none” – turns off dynamic content scaling
- “letterbox” – scales the content up as evenly as possible while still maintaining all of the content on the screen.
- “zoomEven” – preserves aspect ratio while filling the screen uniformly. If the new device has a different aspect ratio, some of the content might be placed off screen.
- “zoomStretch” – scales all content to fill the screen, but doesn’t worry about stretching some of the content vertically or horizontally. All content will remain on the screen.

config.lua file:

```
application =
{
    content =
    {
        width = 320,
```

```
            height = 480,
            scale = "letterbox"
      },
}
```

Dynamic Content Alignment

If you play with dynamic content scaling, you quickly see that there can be issues with alignment. By default, dynamically scaled content evenly divides the additional screen area on all sides of the object for letterbox (as appropriate), and crops the same amount on both sides for zoomEven. In many cases, this won't be a problem. Yet there are times when you will want more control over how alignment occurs for your content. In these cases we will use xAlign and yAlign propreties.
xAlign and yAlign specify the direction of the alignment based upon the x and y axis of the device. xAlign has the possible values of "center" (default), "left", and "right". yAlign has "center" (default), "top", and "bottom".

Dynamic Image Resolution

If we have dynamic scaling and alignment, shouldn't we have dynamic resolution? We do (sort of)! To take full advantage of the higher resolution of newer devices, you will need multiple versions of your graphics. Apple defined a naming convention for developers transitioning to the higher resolution of the iPhone 4 by adding an "@2" suffix to their filenames.
Corona uses a more general method for defining alternative images (Thank you Ansca!) that allows you, the developer, to select your own image naming patterns. The Corona system also does not require you to know the exact resolution of your target device. With the growing field of Android based devices that is a real gift!

To define your image naming convention and the corresponding image resolutions, you will need to create a table named imageSuffix in your config.lua file:

```
application =
{
    content =
    {
        width = 320,
        height = 480,
        scale = "letterbox",
        xAlign = "left",
        yAlign =  "top",

        imageSuffix =
        {
            ["@2"] = 2,
            ["@3"] = 3
        },
    },
```

```
}
```

With this configuration in our config.lua file, we have specified that images with a @2 suffix will be 2 times the base resolution and @3 will be 3 times the base resolution. We could add as many images suffixes as are needed. Now when you go to load your images, use display.newImageRect(“CoolImage.png”, 50, 50) and Corona will choose the closest matching suffix, as defined by your scale.

You can define two additional items with your config.lua file: Frame rate and anti-aliasing: The default frame rate is 30 fps (frames per second). You can go up to 60fps. The fps is locked to the hardware refresh rate on the iPhone (which is 60fps). The fps must divide evenly into 60. This means you have the option of 60fps or 30fps, as anything lower than 30fps is not a priority. To change the default fps from 30 to 60, set fps = 60 in the content area of your config.lua file.

Corona by default has anti-aliasing off. This greatly enhances the system performance and doesn’t have much of an impact on the latest devices. If you find that you need anti-aliasing turned on for your vector objects, you can add the line:

```
antialias = true,
```

to your config.lua file in the content area.

UI.Lua

If you have been looking through some of the sample files provided with your Corona download, you probably came across a file called “ui.lua”. The ui.lua file is available to make button coding actions a little easier and less repetitive in the development process. The ui.lua file adds a button class with labels, font selection, and an event model. At the time of this writing, the current version of ui.lua is version 1.5. However, ui.lua is continuing to evolve very quickly thanks to the community. You can find more information on ui.lua in the Sample Code section of the community.

Using ui.lua, you can easily make a call to create your buttons with more detail. The parameters of newButton through ui.lua include:

- default – the default image for the button
- over – replacement image on a touch
- size – font size for button text
- font – set the font for button text
- text – text that will appear in the button
- emboss – true/false to make the text look embossed
- offset – vertical correction for unusual fonts

Adding Sound

Sound effects and responses are a critical part of any user interface. Sound and music can turn a boring humdrum game or movie into a riveting adventure, if done correctly!
We will be using the new Corona Audio system for all of our projects. The audio system gives us access to advanced OpenAL features and replaces the previous Corona Event Sound system. Corona currently supports up to 32 distinct channels.

Sound File Types

With so many sound file formats available, it is important to select the right formats that will be supported by as many devices as possible. At the time of this writing, supported sound types are:

iOS: .mp3, .caf, .aac, and .wav (16-bit uncompressed)

Android: .mp3, .ogg, and .wav (16-bit uncompressed)

To keep your life simple, plan to use .mp3 and 16-bit uncompressed .wav file formats for all your sound needs. .caf, .aac, and .ogg are great formats but are not accepted by all platforms. So unless you are building for a specific platform and have a special need for one of these file formats, I recommend using mp3 and wav. You should be aware that mp3 does technically have royalty/patent issues. Corona is in the process of adding support for AAC/mp4, which does not have these issues. As you may have noted on the list above, iOS already supports AAC/mp4. Once Android is able to fully support AAC/mp4, I am sure it will be the preferred format for longer sound loops.

Timing Is Everything

The audio system in Corona is a best effort system. It will attempt to play the sound when the request is made. However, if there is a delay (such as a problem with streaming a sound or processor demand), then it will play the sound(s) as soon as it can. This could create a problem in some games or apps, so you should keep it in mind when planning your audio.

Streams and Sounds

There are two ways to load sounds for your app. The first way is to use the audio.loadSound(*filename*) which loads, pre-processes the entire sound file into memory, and be called upon at any time. All of the processing is done on the front end so app performance is not impacted and it can be played on demand:

```
local explosionSound = audio.loadSound( "explosion.wav")
```

The sound can be played as many times as needed using the audio.play() command, with each sound going to a new channel (if needed). For example, if I had a game that had 4 things blown up in a row and each required the sound to be played for explosions, I could issue the commands:

```
audio.play(explosionSound)
audio.play(explosionSound)
audio.play(explosionSound)
audio.play(explosionSound)
```

and each would be played in its own channel. There is no need for the sound to be loaded multiple times; the explosion sound will play multiple times.

audio.loadSound() pre-processes and keeps the sound in ram for quick availability. It is the best solution for small sound files that are regularly used. Be sure to take care of your loading at startup so that the processing of the sound file does not degrade app performance.

The second method to load sounds into your app is with audio.loadStream(). loadStream will load and process small chunks of the sound file as needed. loadStream is best used in situations where possible latency (small slowdowns in app performance) will not have a critical impact upon the usability of the app. Streaming does not use as much memory, so it is considered the best choice for large sound files such as background music.

Unlike loadSound, loadStream can only play one channel at a time. If you needed the same sound file to stream on multiple channels, you would need to load it to two different variables:

```
local backgroundMusic1 = audio.loadStream( "myMusic.mp3")
local backgroundMusic2 = audio.loadStream( "myMusic.mp3")
```

This shouldn't create memory problems since loadStream works with small chunks of memory. However, it could have a performance impact since the sound files are processed in real time.

Through the audio API, we have a great deal of control of the sound elements of our app. We will explore these controls further in chapter 11 when we discuss media in greater detail.

Project 6: Beat-box

Our project will be to create a beat box app to play percussion sounds. For this project you will need to copy the wav and mp3 files (graciously provided for our learning pleasure by Shaun Reed of http://www.constantseas.com!) as well as the ui.lua file from the resource folder into your project folder (which I named BeatBox). Go ahead and create config.lua, build.settings, and main.lua files for this project.

iPhone Beat-Box with dynamic scaling 320x480

iPhone 4 Beat-box with dynamic scaling 640x960

As you can see, thanks to dynamic scaling, the above images are the same, even though the resolution is twice as high on the iPhone 4 as the original iPhone.

config.lua file

```
-- config.lua for project: BeatBox
application =
{
     content =
     {
          width = 320,
          height = 480,
          scale = "letterbox",
          fps = 30,
          antialias = false,
          xAlign = "center",
          yAlign = "center"
     }
}
```

In our config.lua file we have set the default width to 320 pixels, height to 480 pixels with letterbox scaling. The default frames per second will be 30, anti-aliasing is off, and xalign and yalign are set to their default center alignment should scaling be necessary.

build.settings file

```
-- build.settings for project: BeatBox
settings =
{
     orientation =
     {
          default ="landscapeRight",
          supported =
          {
               "landscapeLeft"
          },
     },

}
```

The build.settings file is being used to tell the compiler that this app should be run in landscape mode, with a default to landscapeRight. Portrait is not supported for this app.

In our main.lua file, I am introducing two new commands: system.activate("multitouch") and require("ui"). system.activate("multitouch") is a required command for any app that will be accepting multiple, simultaneous touches. The require('ui") command will load the external file ui.lua so that we will be able make use of the time saving button routines that have been created.

main.lua file

```
-- Project: BeatBox
-- Description: Demonstration app to show dynamic scaling and
playing wav/mp3 sound files
-- Special thanks to Shaun Reed of Constant Seas for providing
the sound files
-- Version: 1.0

system.activate( "multitouch" )     -- allow multi-touch in the
app.

local ui = require("ui")   -- set the variable ui for
referencing ui.lua

----------------------------
```

```
-- load sound files
----------------------------
local snare_wav = audio.loadSound("snare.wav")
local guitar1_wav = audio.loadSound("nylonguitar1.wav")
local guitar2_wav = audio.loadSound("nylonguitar2.wav")
local piano1_wav = audio.loadSound("PianoThingy1.wav")
local piano2_wav = audio.loadSound("PianoThingy2.wav")
local softpiano_mp3 = audio.loadStream("softpianosoundd-
cab.mp3")
```

After setting our system for multi-touch and loading the ui.lua file, we setup variables to load each of the sound files in to. In the last line, softpiano_mp3, we are using streaming instead of load to save memory on our device.

```
-----------------
--  Button Press events
-----------------
local playButton1 = function (event)
     audio.play(snare_wav)
end

local playButton2 = function (event)
     audio.play(guitar1_wav)
end

local playButton3 = function (event)
     audio.play(guitar2_wav)
end

local playButton4 = function (event)
     audio.play(piano1_wav)
end

local playButton5 = function (event)
     audio.play(piano2_wav)
end

local playButton6 = function (event)
     audio.play(softpiano_mp3)
end
```

Next we create the button press events. The events must be declared before we create the buttons, as during the button creation event we specify the event listener. If you wait until after the buttons are created to declare your event, it will not ‘listen’ for the event.

To simplify button creation, I set a variable, w, to hold the value of the display width divided by 5 with an additional 25 pixels removed to center a 50px graphic. This allowed me to evenly space the buttons across the bottom of the device, no matter the number of pixels I was working with, making dynamic scaling much easier.

```
------------------------
-- Create Buttons
------------------------

local w = (display.contentWidth/5) - 25
local snareButton = ui.newButton{
     default = "Button1.png",
     onPress = playButton1,
     text = "Snare",
     size = 12,
     emboss = true
     }

snareButton.x = w
snareButton.y = display.contentHeight - 100

local guitar1Button = ui.newButton{
     default = "Button2.png",
     onPress = playButton2,
     text = "Guitar 1",
     size=12,
     emboss = true
     }

guitar1Button.x = w * 2
guitar1Button.y = display.contentHeight - 100

local guitar2Button = ui.newButton{
     default = "Button3.png",
     onPress = playButton3,
     text = "Guitar 2",
     size = 12,
     emboss = true
     }

guitar2Button.x = w * 3
guitar2Button.y = display.contentHeight -100

local piano1Button = ui.newButton{
     default = "Button4.png",
     onPress = playButton4,
     text = "Piano 1",
```

```
     size = 12,
     emboss = true
     }

piano1Button.x = w * 4
piano1Button.y = display.contentHeight -100

local piano2Button = ui.newButton{
     default = "Button5.png",
     onPress = playButton5,
     text = "Piano 2",
     size = 12,
     emboss = true
     }
piano2Button.x = w * 5
piano2Button.y = display.contentHeight -100

local mp3Button = ui.newButton{
     default = "Button6.png",
     onPress = playButton6,
     text = "Soft Piano",
     size = 12,
     emboss = true
     }
mp3Button.x = display.contentWidth/2 -25
mp3Button.y = display.contentHeight/2
```

You will notice that the simulator does not support multi-touch events (anyone have two mice?). To fully appreciate your composing abilities, you will have to publish the app to your test device.

Summary

In this chapter we began exploring the ability to build for multiple devices, taking advantage of various built-in user interface functions and files that are available with Corona. We also began exploring the various media options that are available for our mobile devices.

Assignments

1) A few additional music loops have been included in the Chapter 6 Figures and Resources folder. Create an app that takes advantage of these additional loops.

2) Add your own sound loops or music to the controller.

3) Change all of the audio load commands to stream instead of load sound.

4) Create a config.lua and build.settings for Assignment 3 in Chapter 5 so that it will run on multiple devices.

5) Create your own sound effects library. Using the UI external library, create an app to play those sounds.

Chapter 7: Application Views

In Chapter 7 we will explore a variety of topics that center around how applications are viewed and flow for the user. This includes:

- Hiding the status bar
- Grouping objects
- Loading external modules and packages
- External Libraries
- Managing multi-view applications

Hiding the Status Bar

Hiding the status bar for an app is a common practice. However, you shouldn't hide the status bar just because you can. Many times the status bar on the smart phone provides important information to the user. If your applications performance or look and feel is not impacted by the status bar, then you should leave it visible. If however the status bar detracts or distracts from the app, then it can be hidden with the command:

```
display.setStatusBar(display.HiddenStatusBar)
```

As a general rule, for most general purpose and information based apps the status bar should remain visible. For game or graphic intensive apps, the status bar should be hidden.

The other options besides display.HiddenStatusBar are:

- display.DefaultStatusBar
- display.TranslucentStatusBar
- display.DarkStatusBar

If you need to know the height of the status bar for calculating placement of objects in your app, the command display.statusBarHeight returns the height in pixels.

Groups

Group objects will quickly become one of your favorite commands for working with multiple display objects. Group allows you to place multiple objects into the same group and be able to apply effects to all of the objects at the same time. This is very handy when working with multiple views and needing to move, fade, or hide a large number of objects quickly. By making a display object a member of a group, you can apply a change to the entire group with just one command.

Think of groups as a basket. Everything that is placed in that basket is moved at the same time, rotated at the same time, can have the color changed at the same time, and can be hidden at the same time.

There are just four commands for working with a group:

- display.newGroup() – creates a new group
- group.numChildren – returns the number of display objects in a group
- group:insert(object) – inserts a new object into a group
- group:remove(index or object) – removes an object from a group

Project 7: Group Movement

In this project we are going to load three images (the buttons from Chapter 6), add them to a group, and use transition.to to move the group down the screen with one command.

main.lua

```
local b1 = display.newImage("Button1.png",10, 10)
local b2 = display.newImage("Button2.png", 100, 50)
local b3 = display.newImage("Button3.png", 200, 100)

local group1 = display.newGroup()
     group1:insert(b1)
     group1:insert(b2)
     group1:insert(b3)

transition.to(group1, {y=300, time=2000})
```

Objects can still be acted upon individually, but I'm sure that you can see how this can be used to easily create the appearance of multiple pages or views of an app without the need to create additional views. Also, an object can only be a member of one group at a time. If you insert it into a second group, it is removed from the first group.

Modules and Packages

As you gain experience creating apps, you will find that certain procedures and code segments are used all the time. Fortunately, Lua allows us to create modules that can be loaded and used in our apps quickly and easily. Shortly, we will begin using a couple of free resources that dramatically reduce our programming time. For now, let's go over how to create your own external library of code.

Project 7.1: External Library

For this project we are going to create a simple library or module that will be called from our main.lua. We will also look at the different ways that external functions can be called. To get started, create a folder. I've named mine Ch7External. This project will not require the build.settings or config.lua files that were discussed in the previous chapter. To create your own external library, simply open a new file with your editor and save as "external.lua" to the folder.

external.lua

```
module(..., package.seeall)


function hiDad()
     textObj = display.newText("Hi Dad", display.contentWidth/2,
display.contentHeight/2, native.systemFont, 24)
     textObj:setTextColor(255,255,255)
end
```

The module(…, package.seeall) is what makes this lua file a module. It is a required command for external files that will be imported into your code later. The function I chose to create as an example is called hiDad (because Mom always gets a shout out, I thought I would give one to Dad this time).

Did you notice anything different from what we did in chapter one on creating our textObj? That's right, there is no 'local' before the variable declaration. Objects that are coming in from an external file must be global instead of local. For our programming purposes, that simply means we don't place the 'local' command in front of the variable. You can (and should) use local variables in your modules if the variable doesn't need to be accessed from outside the module.

Once the module is loaded, there are a couple of ways of accessing their functions. In our main.lua below, I have demonstrated both ways to access our external function hiDad(). First, we must load the external library with the require command:

main.lua

```
local external = require("external")

-- call the external function hiDad() stored in
--external.lua
external.hiDad()

-- cache the external function hiDad() in memory
local hi = external.hiDad

-- call the cached hiDad()
```

```
hi()
```

After an external file is loaded into memory, we can then access any functions that are stored in the library. The first command: external.hiDad() is considered slower, but is more memory efficient. This is the preferred way of accessing an external function if it will not be used multiple times in your app.

In the second way, the function external.hiDad is cached in memory, which will give us much faster access to the function at the cost of a little bit of RAM. The second method is preferred for functions that are regularly used throughout your app.

External Libraries

There are a few external libraries that ship with Corona and two free libraries that I feel warrant mentioning at this point. Looking through the sample apps that come with Corona, you will find a number of libraries:

- ui.lua - (used in chapter 6) simplifies button creation
- movieclip.lua – for assembling animated sprites from separate images
- tableview.lua – simplifies creating tables and lists
- slideview.lua – allows swiping to slide set of images
- scrollview.lua – allows scrolling of text and graphics

Two excellent external libraries that are free for your use are Crawlspace by Adam Buchweitz and Director by Ricardo Rauber.

CrawlSpace

As I mention in discussing resources in Chapter 17, Crawlspace by Adam Buchweitz is the swiss army knife of Corona app development. This collection of routines can dramatically reduce your programming time once you become familiar with all that it can do. You can download crawlspace from Adam's site at http://www.crawlspacegames.com/crawl-space-corona-sdk-library/

Director

Director is a great collection of routines that make multi-view applications quick and easy to create. For large projects that require complex views, the director library has been a wonderful solution. You can download director from the Ansca Mobile site: http://developer.anscamobile.com/code/director-class-10

Project 7.2: Creating a Splash Screen

Typically one of the first things requested by my students is how to add a splash screen to their app. We all know that a good splash screen is critical to any app. It introduces the

app to the user, informs them of who created the app, and gives the hardware a few moments to load any external resources that might be needed.

There are many ways we can add a splash screen. We could create a function in our main.lua to show and dismiss a splash screen. We can use director to handle our splash screen. We could even create the splash screen as an external library that handles animations and preloading of assets. As this is a rather simple project, let's keep the splash screen simple as well, going with the first option of adding the splash screen as a function in our main.lua.

There are also many ways we can develop our splash screen. Usually it will be a png file developed by the artists on your team, but there is nothing keeping you from building a simple screen using text objects and a background.

Starting with the main.lua file from our last project, I have added a function called splash(). The splash function creates a display group to simplify the management of all of the elements that are a part of the splash screen.

main.lua

```
 -- Project: Ch7SplashSreen
local external = require("external")

local function splash()
    -- Create a group to make dismissing the splash screen easy
   splashGroup = display.newGroup()

     -- Create a background with from a vector rectangle.  Must
be a global variable since it is called outside of the function
     bg = display.newRect(splashGroup, 0, 0, 320, 480)
     bg:setFillColor( 10, 10, 200)
```

```
     -- Add text object of app title
     local splashText = display.newText(splashGroup, "Hi\n
Dad!", 100, 150,native.systemFont, 40 )
     splashText.rotation=-30

     -- Tell the user how to proceed
     local proceedText = display.newText(splashGroup, "Tap To
Give A Shout Out", display.contentWidth/2-100,
display.contentHeight-100)

end
```

Using a vector rectangle with a blue fill, the background is added to the splashGroup and also used as our button below. Since it is used outside of the local function, bg must be declared as a global variable.
The splashText is also added to the splashGroup. This text object uses a \n to force a new line in the display and is then rotated -30 degrees, because I liked it better that way.
We will add the proceedText to let the user know what is expected of them, which is always a good user interface consideration.

Next, we add a function to handle when the background is tapped. I chose to fade the splashGroup out over three seconds before calling the main function.

```
local function bgButton(event)
     -- handle dismissing the splash screen when it is tapped by
fading out the splashGroup
     transition.to(splashGroup, {alpha = 0, time = 3000})

     -- pass control to the main function
     main()
end
```

In main I have placed some of the code from our previous project. For this project I am going to build upon Project 7.1: External Library where we give a shout out to Dad. To remove redundancy, I only used the external function caching call. Prior to this call, I added a removeSelf for the splashGroup to remove it from memory. This is always a good practice and will help keep the overhead of your larger projects more manageable.

Finally, we call splash() to get the whole ball rolling and add the event listener for the user to tap the background for the dismissal of the splash screen. Remember, if code is placed in a function, it is not processed until it is called, but it must be made available (or declared) prior to the call.

```
function main()
      -- remove splashGroup from memory
      splashGroup:removeSelf()

      -- cache the external function hiDad() in memory
     local hi = external.hiDad

      -- call the cached hiDad()
     hi()
end

-- now call the splash screen and add the event listener for the
background
splash()
bg:addEventListener("tap", bgButton)
```

Summary

This short but important chapter introduced the concept of using external libraries and files to reduce redundant coding. By making good use of external libraries you can dramatically reduce development time on a project. We also looked at how groups can be used to simplify working with multiple display elements. Finally, we looked at one possible way of implementing a simple splash screen into a project. In the next chapter we will begin to use the built-in physics engine.

Assignments

1) Return to the Beat Box project from Chapter 6 and add a splash screen that utilizes display groups for organization.

2) Examine the crawlspace and director external libraries. Modify Project 7.2 to add the splash screen using one or both of these libraries.

3) Modify the calculator app from chapter 4 by placing the buttons and textfields in a display group.

4) Using groups, create two text groups. One that says “Hi Mom” and one that says “Hi Dad.” Using a button creatd with UI.lua, change between the display groups when the button is tapped.

5) Modify the calculator app from chapter 4 by creating an external library to handle basic math functions.

Chapter 8:
Phun with Physics

The physics in Corona is just plain fun (or phun). In this chapter we will examine the basics of using physics in Corona. This includes:

- Setting gravity
- Types of bodies
- Detecting collisions
- Working with joints

Turn on Physics

The physics implementation in Corona is built upon the popular Box2D. The great people at Ansca have simplified the implementation so that you can quickly and easily add physics to your environment. With just a few lines of code you can add gravity, detect collisions between objects, and use joints to connect objects.
Remember, physics comes at a cost. The number of calculations required by Corona to run your app will dramatically increase. To turn on physics place the commands

```
require("physics")
physics.start(true)
```

at the beginning of your main.lua file. The true parameter that I used is to prevent the bodies that gravity is effecting from going to ‘sleep’. In other words, if a body isn’t involved in a collision, it will go to ‘sleep’, which reduces the overhead on the processor, but in some cases, if the bodies are a sleep, they will stop responding to changes in physics environment.
If it isn’t important that all bodies stay awake, you can use the ‘false’ parameter and save on processor demand.

Scaling

To create accurate pixels to meter ratios, you may need to adjust the scaling of the physics engine. This is only done once before any bodies are added. Scaling can be changed with the command

```
physics.setScale(n)
```

where n should be the width in pixels of the sprite divided by the real-world width. So if an object is 50 pixels on the screen and is 2 meters in the real-world, n should be set to 25 = (50/2). By default, the scale is set to 30 pixels per meter which is optimal to represent 0.1m to 10m objects to correspond to bodies between 3 and 300 pixels in size. This is an

appropriate setting for iPhones through 3GS. For iPhone 4, iPad, and Android devices, you may need to increase this value.

Scaling is based upon original content dimensions. So if you are using the scaling features discussed in previous chapters, you may need to tweak the setScale value to give you more realistic responses.
The setScale property has no impact upon onscreen objects scaling. It only impacts how the physics engine performs calculations.

Bodies

A body is any object that has been changed so that it can simulate a physical object. To make an object a body you use the command physics.addBody(object, [bodyType,] {density=d, friction=f, bounce = b [, radius =r *or* shape=s]})

When you convert a display object into a physics object (a body), the physics engine's rules take over. The physics engine will assume the reference point of the object is the center of the object, no matter where it was set as a display object. Scaling and rotating the object can still be done, but the physics engine will continue to treat the object as it was before the scaling or rotation. So if you are going to scale or rotate, make sure that you do it before you convert it into a physics object. In my experience this means getting any resizing/scaling done before you do your addBody, otherwise it might have some strange results.

Body Types

Body type is an optional string parameter with the possible values of "static", "dynamic", and "kinematic". The default type is "dynamic".

- Static bodies do not move and do not interact with other objects. Typically the ground and walls will be set to static.
- Dynamic bodies are affected by collisions with other objects and gravity.
- Kinematic objects are affected by forces but not by gravity. Draggable objects are set to "kinematic" during the drag event (see Chapter 9 for an example).

Density, Friction, and Bounce

Physical bodies have three main properties; density, friction and bounce.

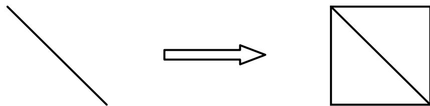
Density is multiplied by the area of the body's shape to determine its mass. The basis of this calculation is that 1.0 is equivalent to water. If a material has less mass than water, such as wood (or a duck or very small rocks - plus 5 points for those who get the reference), the density should be less than 1.0. Heavier materials such as stone or metal will have a density greater than 1.0. But don't feel constrained by these guidelines. It's your app and material can have the density that you feel makes your game flow correctly.

Friction can be any non-negative number. A value of 0 means no friction. A 1.0 is high friction. The default is 0.3. Friction is applied as the body moves through the environment.

Bounce is used to calculate how much of an object's velocity is returned after a collision occurs. A value greater than 0.3 can be considered bouncy. A value of 1.0 would mean that an object keeps all of its velocity; nothing is lost from the collision. A value greater than 1.0 will cause the object to gain velocity after the collision. The default value is 0.2.

Body Shapes

If no shape or radius information is supplied, then the body boundaries will snap to the rectangular boundaries of the display object. While this is fine for a box, the ground, or a platform, it can create strange occurrences if the physics body is a diagonal line shape. In this case, a diagonal line shape will have a bounding box that is a rectangle of the full area between the corners of the object.



Using the default rectangle can simplify calculations, it can also create strange collisions if you have a circle or complex shape. If a radius is provided, then the body boundaries will be circular, centered at the middle of the object used to create the physics body. If a shape is supplied, then the body boundaries will follow the polygon provided by the shape. The maximum sides per shape is 8 and all angles must be convex (angles have to bulge or curve out, not in, or no innie belly buttons, only outies).

When working with complex shapes, you can use polygon bodies to define the shape. If you want to save a great deal of time, I recommend Physics Editor which is discussed in Chapter 17. If you want to do it yourself, then you can set the shape of the object using coordinates. Coordinate sets must be defined in clockwise order with no concave areas:

```
local line = display.newLine(0, 0, 30, 30)
local lineShape = {0, 0, 30, 30}
physics.addBody(line, {density = 2, friction =0.3, bounce=0.3,
shape = lineShape})
```

Body Properties

There are many body properties available to assist your virtual environment to operate correctly. These are all .properties to simplify interacting with them in your app:

- body.isAwake –Boolean. Will fetch the current state of the body or force the body to wake or go to sleep by passing a Boolean. By default, bodies will go to sleep if nothing happens for several seconds until a collision occurs.
- body.isBodyActive – Boolean. Sets or returns the current body. Inactive bodies are not destroyed, but they no longer interact with other bodies.

- body.isBullet – Boolean. Sets whether the body should be treated as a "bullet". Bullets perform continuous collision detection rather than checking on environment (or world) updates. This is processor expensive, but will keep bullets from passing through solid barriers.
- body.isSensor – Boolean. Sets whether the body should be treated as a sensor. Sensors allow other bodies to pass through them, but fire a collision event. Other bodies will not bounce off of a sensor. Sensors do not have to be visible to interact with other bodies.
- body.isSleepingAllowed – Boolean. Sets whether a body is allowed to sleep. The default is true.
- body.isFixedRotation – Boolean. Sets whether a body can rotate. The default is false (i.e. can rotate). Useful for platforms that should not rotate when another object collides or lands on it.
- body.angularVelocity – Number. The value of rotational velocity in degrees per second.
- body.linearDamping – Number. Determines how much a body's linear motion should be dampened. Default is zero.
- body.angularDamping – Number. Determines how much a body's rotation should be dampened. Default is zero.
- body.bodyType – "static", "dynamic", "kinematic". Static bodies do not move and are not affected by other forces (example: the ground). . Dynamic (default) bodies are affected by gravity and collisions. Kinematic bodies are affected by forces other than gravity. Used for drag.

Body Methods

Body methods allow a force to be applied to a body causing it to move or rotate. As with all methods, a ":" is used to separate the object (body) from the method.

- body:setLinearVelocity(x, y) – passes the x and y velocity in pixels per second.
- body:getLinearVelocity – returns the x and y values in pixels per second of the body's velocity. The normal standard command would be: vx, vy = myBody:getLinearVelocity()
- body:applyForce(x, y, body.x, body.y) – applies a linear force or velocity (x, y) to a point in the body. If you apply the force off-center, it will cause the body to spin. Note that the body's density will affect the force required to move the object.
- body:applyTorque(n) – set the applied rotational force. Rotation occurs around its center of mass.
- body:applyLinearImpulse(x, y, body.x, body.y) – A single pulse of force (instead of constant force of applyForce) applied to the object.
- body:applyAngularImpulse(n) – Similar to applyTorque, but is a single pulse of force applied to the object.

Project 8: Using Force

With our first project in chapter 8, we will demonstrate the apply force, torque, linear impulse, and angular impulse methods. I am building this for the droid phone, though it should look fine on anything except an original iPhone.

To begin with, we will load our external libraries, physics and ui, and initialize physics.

```
-- Project: Using Force
-- Description: Demonstrates applying for using various
properties and methods
----------
-- Initialize external libraries and start physics
----------
local ui = require("ui")
local physics = require("physics")
physics.start()
```

Next we will load a simple box image, place it near the center of the device and convert it to a physics body.

```
---------
-- load a box to use for a body
---------
local box = display.newImage("button6.png")
box.x = display.contentWidth/2-100
box.y = display.contentHeight/2-100
physics.addBody(box, {density = 1, friction =0.3, bounce = 0.2})
```

Now that we have everything loaded, we will configure our function for each of the different forces we will apply to the box. Prior to applying the force, we need to return the box to the starting location and make sure that the damping values are returned to zero.

```
-- applyForce demonstration, centered
local boxApplyForce = function (event)
     --return box to its starting location and reset damping
     box.x = display.contentWidth/2-100
      box.y = display.contentHeight/2-100
      box.linearDamping = 0
      box.angularDamping = 0
      --apply force to move up and to the right
      box:applyForce(20, -20, box.x, box.y)

end

-- applyForce demonstration, off-center
local boxApplyForceOffCenter = function (event)
     --return box to its starting location and reset damping
     box.x = display.contentWidth/2-100
      box.y = display.contentHeight/2-100
      box.linearDamping = 0
      box.angularDamping = 0
      -- apply force to move up and to the right, off center
      box:applyForce(20,-20, box.x-20, box.y)

end

-- applyTorque demonstration
local boxApplyTorque = function (event)
     --return box to its starting location and reset damping
     box.x = display.contentWidth/2-100
      box.y = display.contentHeight/2-100
      box.linearDamping = 0
      box.angularDamping = 0
```

```
      -- apply torque
      box:applyTorque(50)
end

-- applyLinearImpulse
local boxApplyLinearImpulse = function (event)
     --return box to its starting location and reset damping
     box.x = display.contentWidth/2-100
      box.y = display.contentHeight/2-100
      box.linearDamping = 0
      box.angularDamping = 0
      --apply a single impulse up and to the right
      box:applyLinearImpulse(20, -20, box.x, box.y)
end

-- applyAngularImpulse
local boxApplyAngularImpulse = function (event)
     --return box to its starting location and reset damping
     box.x = display.contentWidth/2-100
      box.y = display.contentHeight/2-100
      box.linearDamping = 0
      box.angularDamping = 0
      --apply a single angular impulse
      box:applyAngularImpulse(50)
end
```

Our final two functions apply damping effects to the moving box.

```
local boxLinearDamping =  function (event)
     box.linearDamping = 100
end

local boxAngularDamping = function (event)
     box.angularDamping = 100
end
```

Finally, we create each of the buttons using the ui.lua library.

```
---------
-- Create Buttons
---------
local  applyForceButton = ui.newButton{
     default="button1.png",
     onPress=boxApplyForce,
     text= "Force, Centered",
     size = 18,
emboss=true}
     applyForceButton.x = 120
     applyForceButton.y=display.contentHeight - 400
```

```
local  applyForceButtonOffCenter = ui.newButton{
     default="button1.png",
     onPress=boxApplyForceOffCenter,
     text= "Force, off Center",
     size = 18,
     emboss=true }
     applyForceButtonOffCenter.x = 350
     applyForceButtonOffCenter.y=display.contentHeight - 400

local  applyTorqueButton = ui.newButton{
     default="button1.png",
     onPress=boxApplyTorque,
     text= "Torque",
     size = 19,
     emboss=true }
     applyTorqueButton.x = display.contentWidth/2
     applyTorqueButton.y=display.contentHeight - 300

local  applyImpulseButton = ui.newButton{
     default="button1.png",
     onPress=boxApplyLinearImpulse,
     text= "Linear Impulse",
     size = 18,
     emboss=true }
     applyImpulseButton.x = 120
     applyImpulseButton.y=display.contentHeight - 200

local  applyAngularImpulseButton = ui.newButton{
     default="button1.png",
     onPress=boxApplyAngularImpulse,
     text= "Angular Impulse",
     size = 18,
     emboss=true }
     applyAngularImpulseButton.x = 350
     applyAngularImpulseButton.y=display.contentHeight - 200

local  linearDampingButton = ui.newButton{
     default="button1.png",
     onPress=boxLinearDamping,
     text= "Linear Damping",
     size = 18,
     emboss=true }
     linearDampingButton.x = 120
     linearDampingButton.y=display.contentHeight -100
}

local  angularDampingButton = ui.newButton{
```

```
     default="button1.png",
     onPress=boxAngularDamping,
     text= "Angular Damping",
     size = 18,
     emboss=true }
     angularDampingButton.x = 350
     angularDampingButton.y=display.contentHeight -100
```

Gravity

Gravity is very easy to simulate with Corona. You can set gravity for a variety of different effects based upon the x or y direction. A positive value will cause bodies to fall toward the bottom of the screen, while a negative number will cause them to rise toward the top of the screen. If there are no ground or wall bodies (bodies set to static as their type), the bodies being effected by gravity will eventually leave the screen. The default gravity setting is (0, 9.8) which will simulate Earth gravity, pulling bodies downwards on the y-axis.
Gravity is set using

```
physics.setGravity(x, y)
```

To get the current value of gravity, you can use

```
gx, gy = physics.getGravity()
```

Ground and Boundaries

If you are going to keep things from moving off the screen due to physics, you will need to set boundaries. This is done by loading an image and placing it at your boundary. Then when you add the body to physics, set it as a static type.

Your boundary can be anything from a line to a complex sprite environment.

```
local ground = display.newImage("ground.png", 0,320)
physics.addBody(ground, "static")
```

Project 8.1: Playing with Gravity

This is a small little project to demonstrate how gravity can be adjusted and manipulated within your app. I am planning this app for a tablet to give a little more maneuvering room. We will begin by turning on physics and setting gravity to zero. Then create a border area so that our body (a crate from the sample projects) doesn't fall off the screen. Make sure

you use a rectangle for the boundary. Problems can arise if you just use a line. My target devices for this project are the iPad or Galaxy Tab.

```
-- Project: Ch8PlayingWithGravity
local physics = require("physics")
physics.start(true)

-- set inital value for gravity
physics.setGravity(0,0)

-- initialize gx and gy to store gravity changes
gx = 0
gy = 0

-- create border area so object doesn't fall off screen
 local ground = display.newRect(0, 768,  768, 10)
ground:setFillColor(255,255,255,255)

local leftSide = display.newRect(1,1,10,768)
leftSide:setFillColor(255,255,255,255)

local rightSide = display.newRect(758,0,768,768)
rightSide:setFillColor(255,255,255,255)

local top= display.newRect(0,0,768,10)
top:setFillColor(255,255,255,255)

 -- add border to physics as a static object (unaffected by
gravity)
physics.addBody(ground, "static")
physics.addBody(leftSide, "static")
physics.addBody(rightSide, "static")
physics.addBody(top, "static")
```

Next we load an image to be thrown around by the gravity fluctuation.

```
-- load the crate and add it as a body
local crate = display.newImage("crateB.png" )
crate.x = 389
crate.y= 389
physics.addBody(crate, {density=1.0, friction =0.3, bounce =
0.2})
```

By creating a local text object, we can see the current value of gravity. First we create the text objects and load the buttons. Notice that I used the same button image, just rotated it according to the direction that it needs to face.

```
local gravityX = display.newText("0.0",  490, 875,
native.systemFont, 36  )
local gravityY = display.newText("0.0", 195, 875,
native.systemFont, 36 )

-- load arrow buttons and position buttons
local upButton = display.newImage("arrowButton.png", 200, 800)
upButton.rotation = -90

local downButton = display.newImage("arrowButton.png", 200, 950)
downButton.rotation=90

local leftButton = display.newImage("arrowButton.png", 400, 875)
leftButton.rotation=180

local rightButton = display.newImage("arrowButton.png", 600,
875)
```

Using a function, we will update the text object holding the values of vertical and horizontal gravity. Due to the precision of the gravity variable, it is necessary to show just the first 4 digits of the variable. To accomplish this, we will use the sub method; a string method that returns the specific character range you wish to show.

```
-- Update the displayed value of gravity
local function updateGravity()
     gx, gy = physics.getGravity()
     gravityX.text = gx
     gravityX.text = (gravityX.text:sub(1,4))
     gravityY.text  = gy
     gravityY.text = (gravityY.text:sub(1,4))
end
```

Next, we will create a function for each button to handle adjusting the gravity. After adjusting gravity, the text update function is called to update the displayed gravity values.

```
-- adjust the gravity for each button event
local function upButtonEvent (event)
     physics.setGravity(gx,gy-0.1)
     updateGravity()
end

local function downButtonEvent (event)
     physics.setGravity(gx,gy+0.1)
     updateGravity()
end
```

```
local function leftButtonEvent (event)
     physics.setGravity(gx-0.1,gy)
     updateGravity()
end

local function rightButtonEvent (event)
     physics.setGravity(gx+0.1,gy)
     updateGravity()
end
```

And finally, we add our event listeners.

```
-- add event listeners for each button
upButton:addEventListener("tap", upButtonEvent)
downButton:addEventListener("tap", downButtonEvent)
leftButton:addEventListener("tap", leftButtonEvent)
rightButton:addEventListener("tap", rightButtonEvent)
```

As a final note, this project has been supplemented and turned into a game available on the iTunes app store and Google app store.

Collision Detection

If you want to build a game, I am sure you have been wondering, "How do I know when one body hits another?" Three collision events are available through the Corona event listener. The first type is for general collision events and is named "collision". Collision has two phases, "began" and "ended", which represent the initial contact and when contact has ended. These can be used for normal two-body collisions and body-sensor collisions.

The second type of collision event is a "preCollision". This event fires before the objects begin to interact. This type of collision can be very noisy and may send several events prior to actual contact. You should only use preCollision if it is essential to your game logic. Make sure your listener is a local event rather than a global to reduce the overhead and to reduce the number of precollision events.

The final type of collision event is a "postCollision". This event type fires after the objects have interacted. Within this event the collision force is reported and can be used to determine the magnitude of the collision, if needed for your game. The force is returned at the property event.force in a post-collision event. Like precollision, postcollision can be a noisy event generator. It is best to keep the event local and screen out small postcollision forces to maintain your game performance.

Sensors

Sensors are very handy tools in game apps. When another physics body collides with a body that has been turned into a sensor, it fires a collision event. Sensors do not have to be visible and can be any physics body. The difference between a sensor and another body is that a sensor will allow the colliding body to pass through, where as a normal body-body collision will cause a physics reaction, such as bounce, friction, etc.

Sensors are very handy when you need something to begin happening when the player comes within range. It can greatly reduce processing if an animation or other sequence is paused until the player reaches a certain point in the game.

Joints

Joints allow you to join bodies to create complex game objects. To create a joint, you first create the bodies that will be joined. After creating the bodies, you select the type of joint needed to create the effect you desire for your app. The available types of joints are:

- Pivot joint
- Distance joint
- Piston joint
- Friction joint
- Weld joint
- Wheel joint
- Pulley joint
- Touch joint

Pivot Joint

A pivot joint is used to join two bodies that overlap at a point. It can be used in many ways including a ragdoll figure for the head and neck as well as appendages. The initial command to create a pivot joint requires the joint type, the two bodies to be joined, and an anchor point.

```
myNewJoint = physics.newJoint( "pivot", bodyA, bodyB, 200,300 )
```

Each pivot joint has several properties to specify the limitations and actions of the joint:

- .isMotorEnabled(boolean) – allows the pivot point to act as if it had a motor attached. Usually used to simulate a spinning object such as a wheel.
- .motorSpeed(number)- get/sets the linear speed of the motor in pixels per second
- .motorTorque() – returns the torque of the joint motor
- .maxMotorTorque(number) – sets the torque of the joint motor
- .isLimitEnabled(boolean) –get/set whether the joint is limited in motion

- :setRotationLimits(lowerLimit, upperLimit) – sets the rotation limit in degrees from zero.
- :getRotationLimits() – returns the rotation limits in the format `lowerLimit, upperLimit = myNewJoint:getRotationLimits()`
- .jointAngle() – returns the current angle of the joint in degrees.
- .jointSpeed()- returns the speed of the joint in degrees per second.

Distance Joint

Adding a distance joint to your app creates a join between two bodies that are at a fixed distance. The distance should be greater than zero (otherwise, you should use a pivot joint).

```
myNewJoint = physics.newJoint( "distance", bodyA, bodyB,
bodyA.x, bodyA.y, bodyB.x, bodyB.y )
```

The bodyA.x and .y and the bodyB.x and .y are the anchor points for each body. Additional parameters include:

- .length(number) – sets the distance between the anchor points
- .frequency(number) –sets the mass-spring damping frequency in hertz
- .dampingRatio(number) – sets the damping ratio. Range is 0 (no damping) to 1(critical damping).

Piston Joint

The piston joint creates a join between two bodies on a single axis of motion, just like you would expect from a piston or a spring. When creating your bodies for a piston joint, one of them should be dynamic.

```
myNewJoint = physics.newJoint( "piston", bodyA, bodyB, bodyA.x,
bodyA.y, axisDistanceX, axisDistanceY )
```

Unique properties of the piston joint are:

- .jointTranslation() – returns the linear translation of the joint in pixels.
- .jointSpeed() – returns the speed of the joint in degrees per second.

Piston joints may also use the parameters discussed under pivot joint.

Friction Joint

A friction joint is a joint that resists motion, or is ‘sticky’.

```
myJoint = physics.newJoint( "friction", bodyA, bodyB, 200,300 )
```

Its properties are:

- .maxForce(number) – sets the maximum force that can be exerted on the joint.
- .maxTorque(number) – sets the maximum torque that can be applied to the joint.

Weld Joint

Just as the name implies, the weld joint 'welds' two bodies together at a point. It does not allow for movement or rotation.

```
myJoint = physics.newJoint( "weld", bodyA, bodyB, 200,300 )
```

Wheel Joint

A wheel joint combines a piston and pivot joint, acting like a wheel that is mounted on a shock absorber of a car. It makes use of the piston and pivot joint properties.

```
myJoint = physics.newJoint( "wheel", bodyA, bodyB, bodyA.x,
bodyA.y, axisDistanceX,axisDistanceY )
```

Pulley Joint

A pulley joint attaches two bodies with an imaginary line or rope that remains a constant length. If one body is pulled down, the other will move up.
It is more complicated than other joints since it must specify a joint anchor point within each body and a stationary anchor point for the 'rope' to hang from. There is a ratio property associated so that a block and tackle can be simulated (i.e. one side of the rope moves more quickly than the other). By default the ratio is set to 1.0, simulating a simple pulley.

```
myJoint = physics.newJoint( "pulley", bodyA, bodyB, anchorA_x,
anchorA_y, anchorB_x,anchorB_y, bodyA.x, bodyA.y, bodyB.x,
bodyB.y, 1.0 )
```

Read only properties of the pulley joint include:

- .length1() – returns the distance between the 1st joint and the stationary pulley anchor point.
- .length2() – returns the distance between the 2nd joint and the stationary pulley anchor point.
- .ratio() – returns the ratio of the pulley joint

Touch Joint

A touch point creates a temporary elastic joint between a body and your finger. The body will attempt to follow the touch until stopped by other solid objects. If the body that is following the touch collides with another body, a collision event will occur. A body will also rotate based upon gravity when it is 'picked up by an end'.
To move an object by its center point (keeping it from being affected by gravity):

```
touchJoint = physics.newJoint( "touch", crate, crate.x, crate.y
)
```

To move an object based upon where it was touched:

```
touchJoint = physics.newJoint( "touch", crate, event.x, event.y
)
```

Properties of touch joint include:

- .maxForce(number) –get/set the speed of the joint. Default is 1000 for rapid dragging effect.
- .frequency(number) – get/set the frequency of the elastic joint in hertz.
- .dampingRatio(number) – get/set the damping ratio from 0 (no damping) to 1 (critical damping).

Common Methods and Properties for Joints

These properties and methods are available to all joints:

- .getAnchorA() – returns the x, y coordinates of the joints anchor points for bodyA. Values returned are in the local coordinates of the body, so a value of 0, 0 would be the center of the object.
- .getAnchorB() – see .getAnchorA()
- :getReactionForce() – returns the reaction force at the joint anchor for the second body.
- .reactionTorque()- returns the reaction torque at the joint anchor for the second body.
- :removeSelf() – destroys an existing joint and detaches the two bodies.

Project 8.2: Wrecking Ball

This project will demonstrate the use of pivot joints and the impact of density and force upon objects.

We will use four graphic objects for this project: the crane arm, the line (3 copies of it), a ball, and a crate (2 copies). As you might expect, we will begin the project by turning on physics and setting the gravity. I have told the physics engine to not allow items to go to sleep. Usually I avoid this setting; but for everything to react properly, I found it necessary to turn this on. I set the gravity at (0, 9.8) which should simulate Earth gravity for the project.

```
local physics = require("physics")
physics.start(true)
physics.setGravity(0, 9.8)

-- create the ground
 local ground = display.newRect(0, 438,  900, 438)
ground:setFillColor(255,255,255,255)
```

After creating the ground, we load the crane arm, line, wrecking ball, and crates. I found that I need to scale the crane arm up to look right for the app. The lines were placed slightly overlapping so that the pivot joints could be created easily.

```
-- load the crane, line, ball and crate
local crane = display.newImage("crane arm.png", 10, 70)
crane.rotation = 90
crane:scale(2,2)

local line1 = display.newImage("line.png", 170, 20)
local line2 =display.newImage("line.png", 170, 110)
local line3 = display.newImage("line.png", 170, 205)

local ball = display.newImage("wrecking ball.png",110, 280)

local crate1= display.newImage("crateB.png", 300, 300)
local crate2 = display.newImage("crateB.png", 300, 225)
```

Next, add each item as a physics body. The ground and crane will not move and should not be affected by gravity, so we set them as static. The lines should have no bounce or friction, and a higher density than water. The wrecking ball is a 'heavy' item, and needs to be heavier than the crates. I went with 10, which might be a little low, but as you will see

when it comes time to apply force to the ball, we have to keep the density reasonable. Finally I set both crates to a density of 5, keeping the default bounce and friction.

```
-- make all of the objects into phyics bodies
physics.addBody(ground, "static")
physics.addBody(crane, "static")
physics.addBody(line1, {density = 2, friction =0, bounce=0 })
physics.addBody(line2, {density = 2, friction =0, bounce=0 })
physics.addBody(line3, {density = 2, friction =0, bounce=0 })
physics.addBody(ball, {density = 10, friction = 0.7, bounce
=0.2})
physics.addBody(crate1, {density=5})
physics.addBody(crate2, {density=5})
```

Creating the pivot joints is very straight forward. Create a joint for each connection. Select a point where the two items overlap for the final values.
Finally, we apply a linear impulse to the wrecking ball's center of mass to get the ball moving. As you can see, I had to use a fairly high value to get the swing to look the way I wanted.

```
local joint1 = physics.newJoint("pivot", crane, line1, 170, 22)
local joint2 = physics.newJoint("pivot", line1, line2, 170,112)
local joint3 = physics.newJoint("pivot", line2, line3, 170, 206)
local joint4 = physics.newJoint("pivot", line3, ball, 170,280)

ball:applyLinearImpulse(3000, 200, ball.x, ball.y)
```

Trouble Shooting Physics

If you are getting unexpected actions or reactions from your objects, try using the physics.setDrawMode() for troubleshooting help. The setDrawMode property has three settings:

- debug – shows the collision engine outlines of bodies only
- hybrid – overlays collision outlines over the bodies
- normal – default with no collision outlines

```
physics.setDrawMode("debug")
```

Summary

Whew! Let's face it; there are a lot of possibilities when you add physics to the app environment. I'm sure you have many ideas on how you would like to implement some of these tools. In this chapter we looked at how to create a physics-based environment, adding bodies to the environment, applying force to a body, enabling gravity, detecting

collisions, and working with joints. In our next chapter we will use some of these tools to create a game.

Assignments

1) Modify Project 8.1: Playing with Gravity to provide object density, friction and bounce information. Add additional buttons to reset gravity to zero. Adjust the crate density to see the impact on how much gravity is required to move the object.

2) Create a Rube Goldberg Machine that uses the various physics joints and forces to accomplish a very simple action.

3) Modify the density, friction, and bounce of the crates in project 8.2. What impact does changing these parameters have?

4) Add walls inside the gravity area of project 8.1. Modify the bounce settings. Attempt to navigate the box around the new wall. Add a sensor area that will tell you that you have "won" when you move the box to that area.

5) Create 3 simple box bodies and apply force, linear velocity, and linear impulse to each body respectively and observe the results.

Chapter 9: Creating a Game with Corona

In this chapter we are going to build a proof of concept game. The game is entitled "Star Explorer." It is a simple game where you have a space ship with the mission to clear the asteroids out of a sector of space. In this chapter we will examine the concepts of:

- Game design
- Dragging objects
- Collision detection
- Reducing overhead
- A game loop

I originally created this project with one of my mobile programming classes with the beta of the Corona Game Development (now included in the standard release of Corona). The students put so much time and energy into this project; I knew we had something fun, at least from a development standpoint. Several students in that class went on to modify or expand the game for their final project.

Note that this is a proof of concept project. It is not intended to be ready for release to an app store. We will look at what needs to be done to get it ready for the app store in the next chapter.

Game Design

Star Explorer is inspired partially from my misspent youth, spending all the money that I earned in arcade games. After many years playing Asteroids, Galaga, Defender and other popular games of the late 70's, and early 80's, how could I not create my own asteroid shooting game?

The goal for the proof of concept is simple: create a game that has a starship. The starship must shoot the moving asteroids. The ship can be moved by dragging it. It will fire by tapping on the ship. If an asteroid hits the ship, one life is lost. We will start with 3 lives and will keep track of the score for the player as well. I am beginning this project targeting it for the iPhone 3G/GS. In our next chapter we will make the necessary modifications for a device with higher resolution.

As for assets, we will need a ship, asteroid, a graphic for firing, a starry background, and some sound effects.

Again, we will keep everything on one file (main.lua). To begin with, we will hide the status bar and start the physics engine. As we are in space, gravity will be set at 0, 0. I am going to allow the engine the ability to put the bodies that gravity is effecting to sleep in order to reduce the overhead on the processor.

main.lua
```
-- Hide status bar
display.setStatusBar(display.HiddenStatusBar)

-- Setup and start physics
local physics = require("physics")
physics.start()
physics.setGravity(0,0)
```

This is a good place to initialize any variables that will be needed by the game. I usually encourage my students to find a nice big dry-erase board and begin writing down what will be tracked in their game. This is an important step that too many beginning game developers gloss over, assuming they can add the variables as they need. Following that method often leds to repetition of variables and very long debugging sessions.
In considering our variables, try to think through the game process. The obvious variables that we will need are: background, ship, shots fired, asteroids, lives, and score. As we begin to think about the game mechanics a few more variables need to be considered:

- How many shots have been fired? This will help us remove shots that did not collide with anything. We don't want to track shots that are off the screen forever.
- How long ago was the shot fired, or better, what is the maximum age I want for each shot? If it exceeds that limit, it should be removed.
- How many asteroids have been created?
- Asteroids and shots will have to be tracked in an array, due to the number in play. This will simplify checking the age of the shots and if the object has moved off the screen.
- How fast do we want the game to add new asteroids?

- Are we 'dead'? Multiple collisions can create problems with handling the proper number of lives left.

With these concepts in mind, we can begin to initialize our variables. To simplify keeping track of the graphics and sound files for this project, I placed all of my graphics files in a folder named images and my sound files in a folder named sounds.

```
-- Initialize variables
local background = display.newImage ("images/bg1.png", true)
background.x = display.contentWidth /2
background.y = display.contentHeight /2
local lives = 3
local score = 0
local numShot = 0
local shotTable ={}
local asteroidsTable = {}
local numAsteroids = 0
local maxShotAge = 1000
local tick = 200  -- time between game loops in milliseconds
local died=false
```

Now let's setup the Lives and Score text on the display. We will do this with functions so that they can be easily called at the appropriate time during the game. By using functions to set the lives and score text, they will be displayed when we are ready to display them (already starting to think ahead to splash screen and multiple levels).

```
-- Display lives and score
local function newText()
   textLives = display.newText("Lives: "..lives, 10, 30, nil,
12)
   textScore = display.newText("Score: "..score, 10, 10, nil,
12)
   textLives:setTextColor(255,255,255)
   textScore:setTextColor(255,255,255)
end

local function updateText()
    textLives.text = "Lives: "..lives
    textScore.text = "Score: "..score
end
```

Dragging Objects

Let's go ahead and add in the routine for dragging a body. This is an event initiated by a touch on the ship. A touch is different from a tap in that the user continues to touch the ship. This is a standard drag routine used for some drag events. A second more commonly

used drag event can be found in gameUI.lua located in the Multi Puck code sample that comes with Corona. After working with both, I found that I preferred the behavior of this drag event for this particular game. It tracks the bodies' original position, the change to a new position, and changing the body to a kinematic body type so that it can be moved.

```
-- basic dragging physics
local function startDrag( event )
     local t = event.target
     local phase = event.phase
     if "began" == phase then
          display.getCurrentStage():setFocus(t)
          t.isFocus = true

          --Store inital position
          t.x0 = event.x - t.x
          t.y0 = event.y - t.y

          -- make the body type 'kinematic' to avoid gravity
problems
          event.target.bodyType = "kinematic"

          -- stop current motion
          event.target:setLinearVelocity( 0,0)
          event.target.angularVelocity = 0

     elseif t.isFocus then
          if "moved" == phase then
               t.x = event.x - t.x0
               t.y = event.y - t.y0
          elseif "ended" == phase or "cancelled" == phase then
               display.getCurrentStage():setFocus(nil)
               t.isFocus = false

               -- switch body type back to "dynamic"
               if (not event.target.isPlatform) then
                    event.target.bodyType = "dynamic"
               end
          end
     end
     return true
end
```

Next we will load the ship. I set the density, friction, and bounce to the default values. For this version of the game these values do not matter. The final line of code in this section, we give the starfighter a myName value. The myName is used in collision detection so that we know that the starfighter was involved in the collision and can respond appropriately.

```
local function spawnShip()
     starfighter = display.newImage("images/starfighter1.png")
     starfighter.x = display.contentWidth/2
     starfighter.y = display.contentHeight - 50
     physics.addBody (starfighter, {density=1.0, friction = 0.3,
bounce=0})
     starfighter.myName="starfighter"
end
```

Time to load the asteroid body. Each asteroid is set with a low density (1.0) a relatively normal amount of friction, and a rather high bounce rate for better collisions with other asteroids.

There are a couple of things that have to be considered, mainly, where will the asteroid be coming from and moving toward on the screen. To give the game more of a 'moving through space' feel, I decided that asteroids will never appear from the bottom of the screen, only from the sides or top.
First, we need to know how many asteroids have been created so that they can be properly removed later in the game. This is a simple method of garbage collection to keep memory leaks from occurring during the game.
Each asteroid is loaded into our array of asteroids (stored in asteroidsTable). After they are loaded into the array, I used a random number generator to determine which direction the asteroid would load from, left, top, or right.

```
local function loadAsteroid()
     numAsteroids= numAsteroids +1
     asteroidsTable[numAsteroids] =
display.newImage("images/asteroids1-1a.png")
     physics.addBody(asteroidsTable[numAsteroids],{density=1,fri
ction=0.4,bounce=1})
     local whereFrom = math.random(3)--determine direction the
asteroid will appear.
     asteroidsTable[numAsteroids].myName="asteroid"
```

If the asteroid was entering from the left, a random location on the left side was generated, with the bottom 25% 'off limits' for a starting point. After setting its start point, I needed to determine where it was going too. Using transition.to, I generated a random location on the opposite side of the screen and set a random amount of time for the asteroid to move

across the screen. This gives us random fast moving and slow moving asteroids, hopefully making the game more challenging and fun.
The process is then repeated for asteroids entering from the top or the right side.

```
     if(whereFrom==1) then
          asteroidsTable[numAsteroids].x = -50
          asteroidsTable[numAsteroids].y =
(math.random(display.contentHeight *.75))
          transition.to(asteroidsTable[numAsteroids], {x=
(display.contentWidth +100),
          y=(math.random(display.contentHeight)), time
=(math.random(5000, 10000))})
     elseif(whereFrom==2) then
          asteroidsTable[numAsteroids].x =
(math.random(display.contentWidth))
          asteroidsTable[numAsteroids].y = -30
          transition.to(asteroidsTable[numAsteroids], {x=
(math.random(display.contentWidth)),
          y=(display.contentHeight+100), time
=(math.random(5000, 10000))})
     elseif(whereFrom==3) then
          asteroidsTable[numAsteroids].x =
display.contentWidth+50
          asteroidsTable[numAsteroids].y =
(math.random(display.contentHeight *.75))
          transition.to(asteroidsTable[numAsteroids], {x= -100,
          y=(math.random(display.contentHeight)), time
=(math.random(5000, 10000))})
     end

end
```

Collision Detection

On to collision detection! Obviously, this is a fairly important routine, so let's break it down. In the first if then statement, we are looking at the names of the objects that were involved in the collision. If the starfighter was either one of objects, then we check to see if this is the first collision to occur on this starfighter life (it is possible that two collisions occur simultaneously, which causes a double reduction in lives). If this is the first collision (i.e. died == false), then we set died equal to true so that no more collisions are registered until we can handle everything that goes with a starfighter collision.

```
local function onCollision(event)
   if(event.object1.myName =="starfighter" or
event.object2.myName =="starfighter") then
     if(died == false) then
```

```
          died = true
```

Next, we check to see how many lives are left. If the value is 1, then the game is over and an encouraging message is displayed to the player. Otherwise an explosion sound is played using the media API (discussed further in Chapter 11), the starfighter is set to an alpha of 0, lives are reduced by 1, a cleanup routine is called and a routine to re-initialize the starfighter is called with a 2 second delay.

```
          if(lives ==1) then
             media.playEventSound("sounds/explosion.wav")
             event.object1:removeSelf()
             event.object2:removeSelf()
             lives=lives -1
             local lose = display.newText("You Have Failed.", 
30, 150, nil, 36)
             lose:setTextColor(255,255,255)
          else
             media.playEventSound("sounds/explosion.wav")
             starfighter.alpha =0
             lives=lives-1
             cleanup()
             timer.performWithDelay(2000,weDied,1)
          end
     end
   end
```

The second type of collision that is checked for is the collision of an asteroid and a shot fired. Any other type of collision is ignored. During an asteroid and shot collision, the explosion sound is played, the objects are removed and set to a nil value and the score is incremented by 100.

```
   if((event.object1.myName=="asteroid" and 
event.object2.myName=="shot") or (event.object1.myName=="shot" 
and event.object2.myName=="asteroid")) then
          media.playEventSound("sounds/explosion.wav")
          event.object1:removeSelf()
          event.object1.myName=nil
          event.object2:removeSelf()
          event.object2.myName=nil
          score=score+100
   end

end
```

The weDied function moves the starfighter back to its starting position and fades in the ship over 2 seconds. The routine also resets the died variable to false, allowing collisions to occur.

```
function weDied()
     -- fade in the new starfighter
     starfighter.x=display.contentWidth/2
     starfighter.y=display.contentHeight - 50
     transition.to(starfighter, {alpha=1, timer=2000})
     died=false
end
```

Take Your Best Shot

The fireshot function creates and tracks each of the shots fired by the ship. The shot will originate just above the ship, lined up with the current x value of the ships center. Each shot is set as a bullet, forcing the physics engine to check continuously for collision. An age property is used to determine when the shot was fired.

```
local function fireshot(event)
     numShot = numShot+1
     shotTable[numShot] = display.newImage("images/shot.png")
     physics.addBody(shotTable[numShot], {density=1,
friction=0})
     shotTable[numShot].isbullet = true
     shotTable[numShot].x=starfighter.x
     shotTable[numShot].y=starfighter.y -60
     transition.to(shotTable[numShot], {y=-80, time=700})
     media.playEventSound("sounds/fire.wav")
     shotTable[numShot].myName="shot"
     shotTable[numShot].age=0
end
```

Reducing Overhead

The cleanup function removes all asteroids and shots fired from memory each time the player dies. This reduces overhead and frees memory for the next round of play.

```
function cleanup()
     for i=1,table.getn(asteroidsTable) do
          if(asteroidsTable[i].myName~= nil) then
               asteroidsTable[i]:removeSelf()
               asteroidsTable[i].myName=nil
```

```
          end
     end
     for i=1,table.getn(shotTable) do
          if(shotTable[i].myName~= nil) then
               shotTable[i]:removeSelf()
               shotTable[i].myName=nil
          end
     end
end
```

Game Loop

The gameLoop function is the heart of the game. It is called every 200 ms by a timer. It is responsible for updating the Text, loading new Asteroids and removing old shots fired from memory and the screen so that they don't have to be continually processed.

```
local function gameLoop()
     updateText()
     loadAsteroid()
     --remove old shots fired so they don't stack
     for i = 1, table.getn(shotTable) do
          if (shotTable[i].myName ~= nil and shotTable[i].age < 
maxShotAge) then
               shotTable[i].age = shotTable[i].age + tick
          elseif (shotTable[i].myName ~= nil) then
               shotTable[i]:removeSelf()
               shotTable[i].myName=nil
          end
     end
end
```

That takes care of our functions for the game. Remember, functions are not processed until they are called, so to start the game, we need to spawn our first ship, have the text displayed initially, setup our event listeners and a timer. The timer is set to call gameLoop every 200ms (or whatever the tick is set at). To slow or speed up the game, just adjust the tick.

```
--Start the game
spawnShip()
newText()

starfighter:addEventListener("touch", startDrag)
starfighter:addEventListener("tap", fireshot)
Runtime:addEventListener("collision", onCollision)


timer.performWithDelay(tick, gameLoop,0)
```

Time to play test!
Yes, there a lot of things that could be done differently (and with greater memory efficiency), but this project serves as a proof of the game concept. Once you have the concept working correctly, then it is time to make improvements!

Summary

And now you have created your first game for a mobile device! We are far from done with this project. In our next chapter we will look at ways to improve performance, develop it for multiple devices, add a splash screen, and the ability to have multiple levels.

Assignments

1) Modify the project so that the ship rotates in the center of the screen instead of using drag to move the ship.

2) Add additional objects to the space game. It doesn't have to be just asteroids that are being shot.

3) Add additional sound effects for different types of collisions.

4) Background music would be nice! Add a streaming mp3 music track.

Chapter 10:
Star Explorer Continued

With a proof of concept in hand, it is time to turn this simple concept into a full-fledged game. In this chapter we are going to examine how to take our proof of concept game development in the last chapter to something that is ready for the app store. To get this app ready, we will:

- Configure to build for multiple devices
- Add a splash screen
- Make performance improvements
- Configure for playing multiple levels

Configuring the App for Multiple Devices

While it might not be the most exciting thing to do, the first step toward getting Star Explorer ready for market will be to setup the config.lua and build.settings files. I know that you would rather do something fun, like the splash screen. When it comes time to create the splash screen, make performance improvements, and handle multiple levels, it is much easier if everything was already in place to handle multiple resolutions.

My original target device for the game is an iPhone 3G/3GS. So I am setting the base resolution for the game at 320 x 480 and letting the system know that I am supplying graphics for higher resolution devices. I am also limiting the game to run in portrait mode.

build.settings
```
settings =
{
	orientation =
	{
		default ="portrait",
		supported =
		{
			"portrait"
		},
	},

}
```

config.lua
```
application =
{
	content =
	{
```

```
            width = 320,
            height = 480,
            scale = "letterbox",
            fps = 30,
            antialias = false,
            xalign = "center",
            yalign = "center"
      },
      imageSuffix =
      {
            ["@2"] = 2,
       },

}
```

Splash Screen

As we discussed in Chapter 7, splash screens are a very important part of any app. First, they give your app time to do background loading of important elements such as graphics and sound that might otherwise delay the start of the game. Second, a splash screen provides a hint to those who use your app what is going to happen inside the app. Finally, it is a great opportunity for the artistic members of your app development team to really show off!

We are going to use a different method than previously used for this splash screen. Instead of having a function in our main.lua file to load our splash screen, we are going to use an external library for our splash screen. The splash screen is will be composed of several smaller functions, with each function to be called after portion of the object loading is completed. To make the splash screen more interesting, we will animate a portion of the load sequence so that they don't get impatient. As a general rule, the average user will not wait more than 5 seconds for an app to load on their mobile device… unless you give them something to look at to know that good things are coming their way.

Note that this is an example of using several functions and the transition.to command to create a simple animation. While this app is not large enough to need this much load time, it has the benefit of showing how to create such an animation.

The majority of our code will be in the splash.lua file.

As splash.lua will be an external module, we must begin with the module(…, package.seeall) command. To keep the splash screen relatively simple, I broke it into three primary sections, each one lasting 1 to 2 seconds. The calls for each routine are spaced throughout the main.lua. If I needed more loading time, the functions could be broken into smaller parts or more animation could be added.

To begin the splash routine, we will create a local group to place most of the objects in so that they can be faded and removed easily. We will also create and hide the final splash screen that will be used in an event listener at a later point.

You will find that the majority of the variables will be global instead of local. The only time we can use a local variable in our external modules is if the variable will not be used outside the function. As we are primarily using graphics in the splash screen, there are very few local variables in this module.

splash.lua

```
-- Project Ch 10 Star Explorer
-- Splash Screen with action
-- Description: Loads Splash Screen in sections as various parts of
the app load.

module(..., package.seeall)

-- Create a display group to handle dismissing splash screen when
finished.
-- Set to local since it will only be called in splash.lua
local splashGroup = display.newGroup()

fullSplash = display.newImageRect("images/StarExplorerSplash.png",
320, 480)
fullSplash.alpha = 0
```

The first function called will load the background and ship. You will note that the background and ship are both loaded with the newImageRect so that they will properly resize for a larger display than the default iPhone 3GS for which this app was originally designed. They are also placed in the splashGroup for easy dismissal later. The last line of code causes the ship to move toward the center of the screen and scale 50% larger than its standard size. This routine will be called very early in the loading process.

```
-- Load background and ship
function splashBackground()
      splashBg = display.newImageRect(splashGroup,
"images/bg1.png",480, 480 )
      splashBg.x=display.contentWidth/2
      splashBg.y=display.contentHeight/2

      splashShip = display.newImageRect(splashGroup,
"images/starfighter1.png", 74, 80)
      splashShip.x = display.contentWidth/2
      splashShip.y=display.contentHeight-80
      transition.to(splashShip, {y=display.contentHeight/2, xScale=1.5,
yScale=1.5, time  = 1500})
end
```

The second function moves the ship down and to the left (and returning the ship to its original size) while bringing the asteroid into view in the top right of the screen. I was going for a look of maneuvering the ship into the right location to shoot the asteroid.

```
-- Move ship & load asteroid
function splashMoveShip()
      transition.to(splashShip,{x=50, y=display.contentHeight-50,
xScale=1, yScale=1, time=1500})
      splashBigAsteroid = display.newImageRect(splashGroup,
"images/asteroids1.png", 150, 129)
      splashBigAsteroid.x = display.contentWidth+75
      splashBigAsteroid.y = -70
      transition.to(splashBigAsteroid, {x=display.contentWidth - 100,
y=100, rotation = 360, time=6000})
end
```

The splashShoot function handles the remaining action of the splash screen. First, the ship is moved toward the center left of the screen and rotated 45 degrees, toward the asteroid. After the movement, a shoot function is called with a delay of 2 seconds, which will fire a 'shot' at the asteroid. A second function, explosion, is called after 2.2 seconds to simulate the asteroid being blown into small asteroids. The delays are not cumulative. Shoot will perform its actions after a 2 second delay, explosion .2 seconds after shoot. Finally, the full splashscreen function is called 2 seconds after the explosion occurs, for a total of 4.2 seconds for this portion of the splash screen.

```
-- Shoot Asteroid
function splashShoot()
      transition.to (splashShip, {y = display.contentHeight/2, rotation
= 45, time = 1500})
      timer.performWithDelay(2000, shoot )
      timer.performWithDelay(2200, explosion)
end

function shoot()
      shot = display.newImageRect(splashGroup, "images/shot.png", 10,
35)
      shot.x=95
      shot.y=(display.contentHeight/2)-39
      shot.rotation=45
       transition.to(shot, {x=display.contentWidth - 120, y = 120, time
= 300})
       audio.play(fireSound)
end

function explosion()
      splashLittleAsteroid1 = display.newImage(splashGroup,
"images/smallasteroid.png")
      splashLittleAsteroid2 = display.newImage(splashGroup,
"images/smallasteroid.png")
      audio.play(explosionSound)
      splashLittleAsteroid1.x = display.contentWidth -120
```

```
	splashLittleAsteroid1.y = 120
	splashLittleAsteroid2.x = display.contentWidth-120
	splashLittleAsteroid2.y = 120
	transition.to(splashLittleAsteroid1, {x=-20, y = -25, rotation = 
360, time = 1000})
	transition.to(splashLittleAsteroid2, {x = display.contentWidth - 
200, y = display.contentHeight+50, rotation = 360, time=2000})
	splashBigAsteroid.alpha = 0
	shot.alpha = 0
	timer.performWithDelay(2000, splashScreen)
end
```

The splashScreen function hides the splashGroup by setting the alpha to zero, and makes the full splash screen visible by setting its alpha to 1. The splash screen is centered and the text "Tap to Begin" is added. The event listener for the tap event will be in the main.lua file so that the game can be called to start properly. It might seem at first glance that the event listener could be located in the splash.lua file. The problem is that the relationship between main.lua and splash.lua is one way. Main.lua can call a splash.lua function, but a splash.lua cannot call a main.lua function, to start the game.

```
-- Load and display final splash screen
function splashScreen()
	splashGroup.alpha = 0
	fullSplash.alpha = 1
	fullSplash.x = display.contentWidth/2
	fullSplash.y= display.contentHeight/2
	beginText=display.newText("Tap to Begin", display.contentWidth/2-
50, (display.contentHeight*.75),native.systemFont, 24)
end
```

The final function takes care of cleanup for the splash screen. All objects are removed from memory as well as the group. This will free up memory for the game to run as efficiently as possible. The dismissSplashScreen function is called at the beginning of the game, as you will see in the main.lua file.

```
function dismissSplashScreen()
	splashBg:removeSelf()
	splashShip:removeSelf()
	splashBigAsteroid:removeSelf()
	shot:removeSelf()
	splashLittleAsteroid1:removeSelf()
	splashLittleAsteroid2:removeSelf()
	splashGroup:removeSelf()
	fullSplash:removeSelf()
	beginText:removeSelf()
end
```

The changes to main.lua are relatively minor to implement the splash screen. First, we need to load the splash.lua file with 'require'. After the load, we make our first animation call to load the background and ship with the first transition call.

main.lua changes:

```
-- Load external Splash Screen
local splash = require("splash")
splash.splashBackground()
```

Next, about midway through the code, we'll add the second animation call. I made this call with a timer.performWithDelay so that the first animation would have time to complete before the second animation begins.

```
-- Move ship on splash screen and load the asteroid
timer.performWithDelay(1500, splash.splashMoveShip)
```

startGame is a new routine that pulls together several sections of code that were not in functions previously. I added the call to dismiss the splash screen, moved the background initialization out of the variable initialization section at the beginning of main.lua, made the calls to spawn the ship and setup the text, and finally added all the event listeners.

```
function startGame()
      splash.dismissSplashScreen()
      local background = display.newImageRect ("images/bg1.png", 480,
480)
               background.x = display.contentWidth /2
      background.y = display.contentHeight /2
       spawnShip()
      newText()

      starfighter:addEventListener("touch", startDrag)
       starfighter:addEventListener("tap", fireshot)
       Runtime:addEventListener("collision", onCollision)
      timer.performWithDelay(tick, gameLoop,0)
end

-- Shoot the splash asteroid
timer.performWithDelay(3000,  splash.splashShoot)
```

The final bit of code to handle the splash screen is the event listener which calls the new startGame function.

```
splash.fullSplash:addEventListener("tap", startGame  )
```

Not sure where all of this code goes? That's okay. We are getting ready to make some major changes to the main.lua to improve performance. We'll walk through each section and show the changes.

Improving Performance

Unless you have a much better development system than I do, I'm sure you noticed some performance lag in the original version of Star Explorer. With just a few minor changes, we can greatly enhance the performance of the app.

I am including the full main.lua code at this point to ensure comprehension of the changes that are being made. I will note deletions (most of which are actually just moved to a new section) by striking through the text. You can choose to comment out these sections. I will note new code by bolding it. I have also placed some of the functions in a different order to improve the readability from a coder's standpoint.
Beyond adding the splash screen call, the first major change occurs in our variable initialization section.

```
-- Hide status bar
display.setStatusBar(display.HiddenStatusBar)

-- Load external Splash Screen
local splash = require("splash")
splash.splashBackground()

-- Setup and start physics
local physics = require("physics")
physics.start()
physics.setGravity(0,0)
```

In the variable initialization, several changes occurred. First, the background declaration has been moved to the startGame function later in the file. Second, the two sound files, fire.wav and explosion.wav, are being preloaded instead of loaded as needed. This will make one of the greatest differences in our game performance. With the sound preloaded, it only uses memory once and can be called as frequently as needed. Since the sounds are now preloaded, we will have to change how the sound is played later in the app.
Finally we make a second call to the splash.lua file and play the second portion of the animation.

```
-- Initialize variables
local background = display.newImage ("images/bg1.png", true)
background.x = display.contentWidth /2
background.y = display.contentHeight /2
local lives = 3
local score = 0
local numShot = 0
local shotTable ={}
local asteroidsTable = {}
local numAsteroids = 0
local maxShotAge = 1000
local tick = 200  -- time between game loops in milliseconds
local died=false
```

```
fireSound = audio.loadSound("sounds/fire.wav")
explosionSound =  audio.loadSound("sounds/explosion.wav")

-- Move ship on splash screen and load the asteroid
timer.performWithDelay(1500, splash.splashMoveShip)


-- Display lives and score
local function newText()
   textLives = display.newText("Lives: "..lives, 10, 30, nil, 12)
   textScore = display.newText("Score: "..score, 10, 10, nil, 12)
   textLives:setTextColor(255,255,255)
   textScore:setTextColor(255,255,255)
end

local function updateText()
    textLives.text = "Lives: "..lives
    textScore.text = "Score: "..score
end
```

In loadAsteroid, there are only two minor changes to improve performance. When loading the external png file, we now use the newImageRect command which will automatically swap out the asteroid for a higher resolution version should the user be on a higher resolution device. We are now using a new asteroid to reflect the change. The higher resolution asteroid: smallasteroid@2.png will automatically be loaded if needed.

```
local function loadAsteroid()
      numAsteroids= numAsteroids +1
      asteroidsTable[numAsteroids] =
display.newImageRect("images/smallasteroid.png", 31, 26)
      physics.addBody(asteroidsTable[numAsteroids],{density=1,friction=
0.4,bounce=1})
      local whereFrom = math.random(3)
      asteroidsTable[numAsteroids].myName="asteroid"

      if(whereFrom==1) then
            asteroidsTable[numAsteroids].x = -50
            asteroidsTable[numAsteroids].y =
(math.random(display.contentHeight *.75))
            transition.to(asteroidsTable[numAsteroids], {x=
(display.contentWidth +100),
            y=(math.random(display.contentHeight)), time
=(math.random(5000, 10000))})
      elseif(whereFrom==2) then
            asteroidsTable[numAsteroids].x =
(math.random(display.contentWidth))
            asteroidsTable[numAsteroids].y = -30
            transition.to(asteroidsTable[numAsteroids], {x=
(math.random(display.contentWidth)),
            y=(display.contentHeight+100), time =(math.random(5000,
10000))})
      elseif(whereFrom==3) then
```

```
            asteroidsTable[numAsteroids].x = display.contentWidth+50
            asteroidsTable[numAsteroids].y =
(math.random(display.contentHeight *.75))
            transition.to(asteroidsTable[numAsteroids], {x= -100,
            y=(math.random(display.contentHeight)), time
=(math.random(5000, 10000))})
      end
end
```

In the onCollision function, we replace the media.playEventSound with audio.play.

```
local function onCollision(event)
      if(event.object1.myName =="starfighter" or event.object2.myName
=="starfighter") then
            if(died == false) then
                  died = true
                  if(lives ==1) then
                media.playEventSound("sounds/explosion.wav")
                        audio.play(explosionSound)
                        event.object1:removeSelf()
                        event.object2:removeSelf()
                        lives=lives -1
                        local lose = display.newText("You Have Failed.",
30, 150, nil, 36)
                        lose:setTextColor(255,255,255)
                  else
                media.playEventSound("sounds/explosion.wav")
                     audio.play(explosionSound)
                     starfighter.alpha =0
                     lives=lives-1
                     timer.performWithDelay(1000,weDied,1)
                     cleanup()
                  end
            end
      end
      if((event.object1.myName=="asteroid" and
event.object2.myName=="shot") or
         (event.object1.myName=="shot" and
event.object2.myName=="asteroid")) then
           media.playEventSound("sounds/explosion.wav")
            audio.play(explosionSound)
            event.object1:removeSelf()
            event.object1.myName=nil
            event.object2:removeSelf()
            event.object2.myName=nil
            score=score+100
      end
end

function weDied()
      -- fade in the new starfighter
      starfighter.x=display.contentWidth/2
```

```
      starfighter.y=display.contentHeight -50
      transition.to(starfighter, {alpha=1, timer=500})
      died=false
end

function cleanup()
      for i=1,table.getn(asteroidsTable) do
            if(asteroidsTable[i].myName~= nil) then
                  asteroidsTable[i]:removeSelf()
                  asteroidsTable[i].myName=nil
            end
      end
      for i=1,table.getn(shotTable) do
            if(shotTable[i].myName~= nil) then
                  shotTable[i]:removeSelf()
                  shotTable[i].myName=nil
            end
      end
end
```

The only changes in spawnShip and fireshot are to use the newImageRect for loading the ship and replace media.playEventSound with audio.play.

```
local function spawnShip()
      starfighter = display.newImageRect("images/starfighter1.png", 74,
80)
      starfighter.x = display.contentWidth/2
      starfighter.y = display.contentHeight - 50
      physics.addBody (starfighter, {density=1.0, friction = 0.3,
bounce=1.0})
      starfighter.myName="starfighter"
end

local function fireshot(event)
      numShot = numShot+1
      shotTable[numShot] = display.newImageRect("images/shot.png", 10,
35)
      physics.addBody(shotTable[numShot], {density=1, friction=0})
      shotTable[numShot].isbullet = true
      shotTable[numShot].x=starfighter.x
      shotTable[numShot].y=starfighter.y -60
      transition.to(shotTable[numShot], {y=-80, time=700})
       media.playEventSound("sounds/fire.wav")
      audio.play(fireSound)
      shotTable[numShot].myName="shot"
      shotTable[numShot].age=0
end


-- basic dragging physics
local function startDrag( event )
      local t = event.target
```

```
	local phase = event.phase
	if "began" == phase then
		display.getCurrentStage():setFocus(t)
		t.isFocus = true

		--Store inital position
		t.x0 = event.x - t.x
		t.y0 = event.y - t.y

		-- make the body type 'kinematic' to avoid gravity problems
		event.target.bodyType = "kinematic"

		-- stop current motion
		event.target:setLinearVelocity( 0,0)
		event.target.angularVelocity = 0

	elseif t.isFocus then
		if "moved" == phase then
			t.x = event.x - t.x0
			t.y = event.y - t.y0
		elseif "ended" == phase or "cancelled" == phase then
			display.getCurrentStage():setFocus(nil)
			t.isFocus = false

			-- switch body type back to "dynamic"
			if (not event.target.isPlatform) then
				event.target.bodyType = "dynamic"
			end
		end
	end
	return true
end

local function gameLoop()
	updateText()
	loadAsteroid()
	--remove old shots fired so they don't stack
	for i = 1, table.getn(shotTable) do
		if (shotTable[i].myName ~= nil and shotTable[i].age < 
maxShotAge) then
			shotTable[i].age = shotTable[i].age + tick
		elseif (shotTable[i].myName ~= nil) then
			shotTable[i]:removeSelf()
			shotTable[i].myName=nil
		end
	end
end
```

The startGame function begins by calling the dismiss splash screen function in splash.lua. Next, the function loads the background and positions it. This section of code was moved from the variable initialization area so that it would not load over the splash screen.

SpawnShip and newText were moved from their previous locations at the end of the main.lua file to this section so that start game now handles on beginning functions, including the declaration of the event listeners and the timer.

```
function startGame()
      splash.dismissSplashScreen()
    local background = display.newImageRect ("images/bg1.png", 480,
480)
    background.x = display.contentWidth /2
    background.y = display.contentHeight /2
    spawnShip()
    newText()

    starfighter:addEventListener("touch", startDrag)
    starfighter:addEventListener("tap", fireshot)
    Runtime:addEventListener("collision", onCollision)
    timer.performWithDelay(tick, gameLoop,0)
end
```

The final changes to main.lua include the last call to the splash.lua file to handle the final animation and adding an event listener for when the user is ready to begin the game. It was necessary to delay the call to the splash screen due to the size of this program. A larger, more sophisticated game might not need that much of a delay. However, timed delays do give you a consistent startup without being impacted by newer equipment that might take care of the preloading much faster than your animations are ready. It requires careful consideration when planning startup delays.

```
-- Shoot the splash asteroid
timer.performWithDelay(3000,  splash.splashShoot)

splash.fullSplash:addEventListener("tap", startGame  )

--Start the game
spawnShip()
newText()

starfighter:addEventListener("touch", startDrag)
starfighter:addEventListener("tap", fireshot)
Runtime:addEventListener("collision", onCollision)

timer.performWithDelay(tick, gameLoop,0)
```

Varying Difficulty

Now we have improved performance and a splash screen sequence. This app is almost ready for the big time! To make the game more interesting, we are going to add two more features: different sized asteroids and faster game play as the player increases in score.

Increasing Game Speed

An easy adjustment to make is the tick. In our initialization area, change the tick from 200 to 400. This will slow how quickly asteroids are introduced to the environment.
Now, in the gameLoop function we will add an if then sequence to adjust the tick based upon the players score:

```
local function gameLoop()
      updateText()
      loadAsteroid()
      --remove old shots fired so they don't stack
      for i = 1, table.getn(shotTable) do
            if (shotTable[i].myName ~= nil and shotTable[i].age <
maxShotAge) then
                  shotTable[i].age = shotTable[i].age + tick
            elseif (shotTable[i].myName ~= nil) then
                  shotTable[i]:removeSelf()
                  shotTable[i].myName=nil
            end
      end

      if score > 2000 and tick >350 then
            tick = 350
      elseif score > 5000 and tick > 300 then
            tick = 300
      elseif score> 10000 and tick > 250 then
            tick = 250
      elseif score > 15000 and tick > 200 then
            tick = 200
      elseif score > 20000 and tick > 150 then
            tick = 150
      elseif score > 25000 and tick > 100 then
           tick = 100
      end

end
```

A Little Variety

To make the game more visually appealing, we can add more asteroids of varying sizes. In the resource folder for chapter 10 you will find several different size and shapes of asteroids that can be used for this project.
Editing the loadAsteroid function, we can add a random number generator to make the environment more challenging and random. In the example below, we have added the possibility of three different asteroids, each being a different size.

```
local function loadAsteroid()
      numAsteroids= numAsteroids +1

   -- Randomly select an asteroid type
   local randomAsteroid = math.random(3)

   if randomAsteroid == 1 then
       asteroidsTable[numAsteroids] = display.newImageRect(
"images/smallasteroid.png", 31, 26)
   elseif randomAsteroid == 2 then
        asteroidsTable[numAsteroids] = display.newImageRect(
"images/asteroids1.png", 150, 129 )
   elseif randomAsteroid == 3 then
        asteroidsTable[numAsteroids] = display.newImageRect(
"images/asteroids2.png", 200, 136)
   end

  physics.addBody(asteroidsTable[numAsteroids], {density=1,
friction=0.4,bounce=1})
  local whereFrom = math.random(3)
```

Of course there is much more that we could change and add. The larger asteroids could break into smaller asteroids as in our splash screen, there could be aliens flying through. We could add a 'boss' asteroid… etc.

Summary

In this chapter we have looked at how to include a second type of splash screen that entertains the user while the game loads. We also examined how to improve performance and delay performance of functions with the timer.performWithDelay command. Finally, we examined how to vary the game difficulty by adjusting the speed and varying the targets.

Assignments

1) Upgrade the app by adding more types of asteroids and causing the asteroids to spin in different directions and at varying speeds.

2) Modify the collision routine so that larger asteroids spawn smaller asteroids. For an additional challenge, make the smaller asteroids worth more points.

3) Add a Shield capability to the starship.

4) Add an enemy alien race to attack the starship.

5) Change the graphics to make the game completely different. Maybe it should be hamburgers instead of asteroids being eaten.

Chapter 11: Media Makes the World Go Round (or Can You Hear Me Now?)

The different types of media available for smart phones and tablets inspire many different types of software. In this chapter we will look at the various tools associated with different types of media, including:

- The various types of sound APIs
- Different ways to play and record sound
- Playing movies
- Working with streaming video
- Working with files in different folders
- Using the built in Camera

Working with Sound

We have done a few apps that incorporate sound thus far. Let's take a closer look at the sound APIs that are available. As you may have noticed, we have a couple of different ways of working with sound in Corona. This is due to Corona's audio system being powered by OpenAL. Formerly Corona used an event sound vs. non-event sound system through the media library. This is being replaced by the OpenAL system through the audio API. While the media.playEventSound and media.playSound still work (as evidenced by their use in Chapter 9), you should use the audio API for all sound events (used in chapters 6 and 10).

As Corona takes advantage of OpenAL, we have a great deal of power and flexibility available for our sound applications. To simplify the 30 different audio properties and methods, let's categorize them into Basic controls, Duration Controls, Volume Controls, and Channels.

Basic Audio Controls

The basic audio controls provide the foundation of working with sound files, allowing the loading, playing, and stopping of sound files. The basic properties are:

- audio.loadSound(*filename*) – Loads the entire sound file into memory.
- audio.loadStream(*filename*) – Opens a file to read as a stream.
- audio.play(*audioHandle, {[*channel=*c] [,* loops=*1] [*duration *=d] [,* fadein=*f] [,* onComplete=*o] }*) – begins the play of the previously loaded audio loop (either via loadSound or loadStream). All additional parameters are optional. Channel will assign the audio playback to a specific channel (auto selected if omitted); loops sets the number of times the playback will loop (default 0); duration will stop playback

at a specific time, whether the audio file is finished playing or not, in milliseconds; fadein controls how long to take to increase the playback to full volume in milliseconds; onComplete passes an event parameter back to the calling procedure on the completion of the playback. Options to be returned include: channel, handle (audio variable), or completed (true if normal completion, false if audio was stopped).

- audio.pause([*audioHandle*]) - pauses playback on specified channel or all channels if no parameters are included.
- audio.resume([*audioHandle*]) – resumes playback on specified channel or all channels if no parameters are included.
- audio.stop(([*audioHandle*]) - stops playback on specified channel or all channels if no parameters are included.
- audio.stopWithDelay(duration [, { *audioHandle* }]) - stops playback on specified channel or all channels if no parameters are included after the given number of milliseconds.
- audio.rewind*([audioHandle] [, { channel=c }*]) – rewinds the specified audio to its beginning position. For files loaded with audio.loadSound, you may only rewind based upon channel, not handle (since multiple instances of the audio handle could be playing). audio.loadStream can be called by handle or channel, but may not update until after the current buffer finishes playing. To rewind 'instantly', stop the stream, rewind, and then play.
- audio.seek(*time* [, *audioHandle*] [, {*channel = c*}]) – Seeks to a time position in the audio file. If no handle or channel is specified, all audio will seek to the specified time, which is provided in milliseconds.
- audio.dispose(*audioHandle*) – release memory that was associated with the handle. The audio should not be active when it is freed.

As mentioned in Chapter 6, files loaded with audio.loadSound can be shared and played multiple times across different channels. audio.loadStream cannot be shared. To play multiple streams, they must be loaded separately.

Duration Audio Controls

- audio.fade([{ [channel = c] [, time=t] [, volume=v] }]) – fades a playing sound in the specified time to the specified volume. If channel is not specified, all channels fade. If time is omitted, the default fade time is 1000 milliseconds. Volume may be range from 0.0 to 1.0. If omitted, the default value is 0.0.
- audio.fadeOut([{ [channel = c] [, time=t] }]) – stops the playing sound in a specified amount of time and fades to a minimum volume. At the end of the time, the audio will stop and release the channel. To fadeout all channels, specify 0. Time default is 1000 milliseconds.
- audio.getDuration(*audioHandle*) – returns the total time in milliseconds of the audio. If the length cannot be determined, -1 is returned.

Volume Controls

- audio.setVolume(volume[, { [channel = c] }]) – sets the volume of a specified channel or the master volume if no channel is specified. Volume may range from 0.0 to 1.0
- audio.setMaxVolume(volume[, { [channel = c] }]) – sets the maximum volume for all channels.
- audio.setMinVolume(volume[, { [channel = c] }]) - sets the minimum volume for all channels.

- audio.getVolume([{ [channel = c] }]) – returns the volume of the channel of master volume if no channel is specified.
- audio.getMaxVolume({ channel = c}) - returns the maximum volume of a channel. Returns average maximum volume if no channel is specified.
- audio.getMinVolume({ channel = c}) - returns the minimum volume of a channel. Returns average minimum volume if no channel is specified.

Audio Channels

Corona uses a channel system to keep track of various sounds that are playing within your app. At this time there are 32 channels available for audio playback.

- audio.findFreeChannel([startChannel]) - returns the channel number of an available channel or 0 if no channels are available.
- audio.freeChannels – returns the number of channels that are available.
- audio.isChannelActive(channel) – returns true if specified channel is playing or paused.
- audio.isChannelPaused(channel) – returns true if specified channel is paused.
- audio.isChannelPlaying(channel) – returns true if specified channel is playing.
- audio.reserveChannels(channels) – reserves a certain number of channels so they will not be automatically assigned to play calls. Typically used to reserve lower number channels for background music, voice over, or specific sounds. 0 will unreserve all channels. A number between 1 and 32 set aside the specified number of channels.
- audio.reservedChannels – returns the number of reserved channels
- audio.totalChannels – returns the total number of channels (currently 32).
- audio.unreservedFreeChannels – returns the number of channels available for playback, excluding reserved channels.
- audio.unreservedUsedChannels – returns the number of channels in use excluding reserved channels.
- audio.usedChannels – returns the number of channels in use including reserved channels.

Sound File Types (Revisited)

As one of the major reasons why people adopt Corona is the ability to build for multiple platforms, we must keep in mind which sound file types are available for use with both platforms. The supported sound file types are:

iOS: .mp3, .caf, .aac, and .wav (16-bit uncompressed)

Android: .mp3, .ogg, and .wav (16-bit uncompressed)

To keep your life simple, plan to use .mp3 and 16-bit uncompressed .wav file formats for all your sound needs. .caf, .aac, and .ogg are great formats but are not accepted by all platforms. So unless you are building for a specific platform and have a special need for one of these file formats, I recommend using mp3 and wav. You should be aware that mp3 does technically have royalty/patent issues. Corona is in the process of adding support for AAC/mp4, which does not have these issues. As you may have noted on the list above, iOS already supports AAC/mp4. Once Android is able to fully support AAC/mp4, I am sure it will be the preferred format for longer sound loops.

Where did I put that file?

As we begin to discuss recording and accessing external files such as media files, photos, and in our next chapter, databases, let us take a moment to discuss the directories on your device.

- system.pathForFile(*filename [, baseDirectory]*) – provides an absolute path to access files. Returns nil if file does not exist.
- system.DocumentsDirectory – should be used for files that need to persist between sessions. When used in the simulator, the user's documents directory is used.
- system.ResourceDirectory – is the folder or directory where your assets are stored. Do not change anything in this folder while the app is running. It could invalidate the app and the OS will consider the app malware and refuse to launch. system.ResourceDirectory is assumed (default) when loading assets for your app.
- system.TemporaryDirectory – Just as the name says, is temporary. Only use for in app data. No guarantee that the file will be there the next time the app is used.

Multimedia API

The multimedia API is used for video, camera, and photo library on your device. Originally it was also used for audio, but those APIs are in the process of being replaced with the appropriate audio property or method. Depreciated (i.e. don't use these) media APIs include:

- media.getSoundVolume – depreciated. Use audio.getVolume API.

- media.newEventSound() – depreciated. Use audio.loadSound API.
- media.pauseSound()– depreciated. Use audio.pause API.
- media.playEventSound()– depreciated. Use audio.play API.
- media.playSound()– depreciated. Use audio.play API.
- media.setSoundVolume()– depreciated. Use audio.setVolume API.
- media.stopSound()– depreciated. Use audio.stop API.

Recording Audio

It is possible to record audio using the Corona interface. While different platforms support different formats, both Apple and Android support raw.

- media.newRecording([*path*]) – Creates the object for audio recording. If the path is omitted, the recorded audio will not be saved.
- object:startRecording() – starts audio recording and cancels any audio playback.
- object:isRecording() – returns true if audio recording is in progress.
- object:stopRecording() – stops audio recording.
- object:setSampleRate(r) – sets the sampling rate. Valid rates are: 8000, 11025, 16000, 22050, and 44100. Note: Windows simulator with a sample rate of 44100Hz may create an aif file that is corrupt and not playable. setSampleRate must be called before the startTuner.
- object:getSampleRate() – returns the current audio recording sample rate.
- object:startTuner( ) – Turns on audio tuning feature. Should be started before startRecording is called.
- object:stopTuner() – stops the tuner.
- object:getTunerFrequency() – returns the last calculated frequency in Hz.
- object:getTunerVolume() – returns the mean squared normalized sample value of the current audio buffer (i.e., a value between -1 & 1).

Project 11: Simple Audio Recorder

Ansca Mobile has been kind enough to include several sample projects using the various media and audio APIs. For this project walk-through, rather than re-inventing the wheel, we will look at the logic behind the Simple Audio Recorder (which can be found in the Sample Code/Media folder or under Ch11SimpleAudioRecorder).

build.settings

```
settings =
{
   androidPermissions =
   {
       "android.permission.RECORD_AUDIO"
   },
```

```
      iphone =
      {
            plist =
            {
               CFBundleIconFile = "Icon.png",
               CFBundleIconFiles = {
                  "Icon.png",
                  "Icon@2x.png",
                  "Icon-72.png",
               },
            },
      }
}
```

The build.settings has one new command, permission for Android devices to allow audio recording.

config.lua

```
application =
{
        content =
        {
                width = 320,
                height = 480,
                scale = "letterbox"
        },
}
```

Nothing that we haven't used before in the config.lua file, so on to the main.lua.

To begin with, I have copied a portion of the comment declaration showing that this code is shared under the MIT license. The first few lines load ui.lua for handling buttons, a background image and position the background in the center of the screen.

main.lua

```
-- Sample code is MIT licensed, see
-- http://developer.anscamobile.com/code/license
-- Copyright (C) 2010 ANSCA Inc. All Rights Reserved.
-----------------------------------------------------------

-- Load external button/label library (ui.lua should be in -- the same folder as main.lua)
local ui = require("ui")
local background = display.newImage("carbonfiber.jpg", true)
background.x = display.contentWidth/2
background.y = display.contentHeight/2
```

Next, we set the file name where the audio file will be recorded. In this example, it is being recorded as an ".aif" file type, which is concatenated onto the file name if the app is being run in the simulator or on an iPhone. If the app is deployed to an Android, it will be saved as ".pcm" file type.

```
local dataFileName = "testfile"
if "simulator" == system.getInfo("environment") then
    dataFileName = dataFileName .. ".aif"
else
    local platformName = system.getInfo( "platformName" )
    if "iPhone OS" == platformName then
        dataFileName = dataFileName .. ".aif"
    elseif "Android" == platformName then
        dataFileName = dataFileName .. ".pcm"
    else
     print("Unknown OS " .. platformName )
    end
end
print (dataFileName)
```

Setup three labels for display of text to the screen and a record button.

```
-- title
local t1 = ui.newLabel{
     bounds = { 10, 25, 300, 40 },
     text = "Echo",
     font = "AmericanTypewriter-Bold",
     textColor = { 255, 204, 102, 255 },
     size = 40,
     align = "center"
}
local t2 = ui.newLabel{
     bounds = { 10, 75, 300, 40 },
     text = "Simple Audio Recorder",
     font = "AmericanTypewriter-Bold",
     textColor = { 255, 204, 102, 255 },
     size = 22,
     align = "center"
}

--  recording status area
local roundedRect = display.newRoundedRect( 10, 160, 300, 80, 8 )
roundedRect:setFillColor( 0, 0, 0, 170 )
local s = ui.newLabel{
     bounds = { 10, 180, 300, 40 },
     text = " ",
     textColor = { 10, 240, 102, 255 },
     size = 32,
     align = "center"
}

-- sampling rate display
local s2 = ui.newLabel{
     bounds = { 10, 380, 300, 40 },
     text = " ",
     textColor = { 130, 200, 255, 255 },
     size = 14,
```

```
      align = "center"
}
```

Create variables for recording, the button, and playback state. The variable r will be used as the recording object.

```
local r                 -- media object for audio recording
local recButton         -- gui buttons
local fSoundPlaying = false   -- sound playback state
local fSoundPaused = false    -- sound pause state

-- Sets the text of a button
local function setButtonText( button, text )
      button[3].text  = text      -- Highlight
      button[4].text  = text      -- Shadow
      button[5].text  = text
end
```

To get the most use out of the space we have on the screen, the text of the button will be dynamically changed based upon the status of the recording. In this section of code, the isRecording method is used to check on the status of the recording. If recording has been completed, then the option of pausing playback and resuming playback are offered to app user.

```
-- Update the state dependent texts
local function updateStatus ()
    local statusText = " "
    local statusText2 = " "
    if r then
        local recString = ""
        local fRecording = r:isRecording ()
        if fRecording then
            recString = "RECORDING"
            setButtonText (recButton, "Stop recording")
        elseif fSoundPlaying then
            recString = "Playing"
            setButtonText (recButton, "Pause playback")
        elseif fSoundPaused then
            recString = "Paused"
            setButtonText (recButton, "Resume playback")
        else
            recString = "Idle"
            setButtonText (recButton, "Record")
        end
```

The final portion of the if..then sequence updates the sampling rate if recording is active. This uses the r:getSampleRate() method.

```
        statusText =  recString
        statusText2 = "Sampling rate: " .. tostring (r:getSampleRate()
.. "Hz")
```

```
    end
    s:setText (statusText)
    s2:setText (statusText2)
end
```

Now a few event handlers to handle when the sound has finished playback (freeing up memory) and button press events.

```
-----------------------------------------------------------
--  *** Event Handlers ***
-----------------------------------------------------------
local function onCompleteSound (event)
    fSoundPlaying = false
    fSoundPaused = false
     -- Free the audio memory and close the file now that we are done
playing it.
     audio.dispose(event.handle)
    updateStatus ()
end

local function recButtonPress ( event )
    if fSoundPlaying then
        fSoundPlaying = false
        fSoundPaused = true
           audio.pause() -- pause all channels
    elseif fSoundPaused then
        fSoundPlaying = true
        fSoundPaused = false
           audio.resume() -- resume all channels
    else
```

In this next section of code, the playback function is handled. The filepath is set to play the file that has been saved to the documents directory so that it will continue to exist after the app closes. Notice that the media.playSound has been replaced by audio.loadStream.

```
        if r then
            if r:isRecording () then
                r:stopRecording()
                local filePath = system.pathForFile( dataFileName,
system.DocumentsDirectory )
                -- Play back the recording
                local file = io.open( filePath, "r" )
                if file then
                    io.close( file )
                    fSoundPlaying = true
                    fSoundPaused = false
                    --media.playSound( dataFileName,
system.DocumentsDirectory, onCompleteSound )
                    playbackSoundHandle = audio.loadStream(
dataFileName, system.DocumentsDirectory )
                           audio.play( playbackSoundHandle, {
onComplete=onCompleteSound } )
```

```
                end
            else
                fSoundPlaying = false
                fSoundPaused = false
                r:startRecording()
            end
        end
    end
    updateStatus ()
end
```

This section tests for the highest allowed sample rate that the device will allow when the user attempts to increase the sampling rate. First, an array of potentially supported sampling rates is created (theRates). Next a while do loop is used to step through the rates, comparing the rates to f, which contains the getSampleRate data. Only sample rates that are compatible with the device will be allowed.

```
-- Increase the sample rate if possible
-- Valid rates are 8000, 11025, 16000, 22050, 44100 but
-- many devices do not support all rates
local rateUpButtonPress = function( event )
    local theRates = {8000, 11025, 16000, 22050, 44100}
    if not r:isRecording () and not fSoundPlaying and not fSoundPaused
then
--        r:stopTuner()
        local f = r:getSampleRate()
        --  get next higher legal sampling rate
        local i, v = next (theRates, nil)
        while i do
                if v <= f then
                    i, v = next (theRates, i)
                else
                    i = nil
                end
        end
        if v then
            r:setSampleRate(v)
        else
            r:setSampleRate(theRates[1])
        end
--        r:startTuner()
    end
    updateStatus()
end
```

Much like the previous section, this function handles the user requesting to lower the sampling rate.

```
-- Decrease the sample rate if possible
-- Valid rates are 8000, 11025, 16000, 22050, 44100 but ----- many
devices do not support all rates.
```

```
local rateDownButtonPress = function( event )
    local theRates = {44100, 22050, 16000, 11025, 8000}
    if not r:isRecording () and not fSoundPlaying and not fSoundPaused
then
--        r:stopTuner()
        local f = r:getSampleRate()
        --  get next lower legal sampling rate
        local i, v = next (theRates, nil)
        while i do
                if v >= f then
                    i, v = next (theRates, i)
                else
                    i = nil
                end
        end
        if v then
            r:setSampleRate(v)
        else
            r:setSampleRate(theRates[1])
        end
--        r:startTuner()
    end
    updateStatus()
end
```

Finally, buttons are created and positioned on the screen.

```
-------------------------------------------
-- *** Create Buttons ***
-------------------------------------------

-- Record Button
recButton = ui.newButton{
     default = "buttonRed.png",
     over = "buttonRedOver.png",
     onPress = recButtonPress,
     text = "Record",
     emboss = true
}

--  increase sampling rate
local rateUpButton = ui.newButton{
     default = "buttonArrowUp.png",        -- small arrow image
     over = "buttonArrowUpOver.png",
     onPress = rateUpButtonPress,
     id = "arrowUp"
}

-- decrease sampling rate
local rateDownButton = ui.newButton{
     default = "buttonArrowDwn.png",       -- small arrow image
     over = "buttonArrowDwnOver.png",
```

```
      onPress = rateDownButtonPress,
            id = "arrowDwn"
}

-----------------------------------------------
-- *** Locate the buttons on the screen ***
-----------------------------------------------
recButton.x = 160;            recButton.y = 290
rateUpButton.x =   190;    rateUpButton.y =   420
rateDownButton.x = 140;     rateDownButton.y = 420
```

This last bit of code gets everything started for the app by setting a file path to record the audio to a file in the documents area of the device and assigning r to the new recording.

```
local filePath = system.pathForFile( dataFileName,
system.DocumentsDirectory )
r = media.newRecording(filePath)
updateStatus ()
```

Video Playback

Yes, you can play video through Corona. When you call for video playback the media player interface takes over. If showControls is true, the user can pause, start, stop and seek in the video. It is a good idea to use a listener to notified your app when the video has ended. iOS supported formats include .mov, .mp4, .m4v, and .3gp using H264 compression at 640x480 at 30fps and MPEG-4 Part 2 video.
Android playback is not yet supported through Corona.

- media.playVideo(*path [,baseSource], showControlds, listener*) – plays the video in a device-specific popup video media player.

Camera

The final piece of the media tools is the camera. The API call opens a platform-specific interface to the device camera or photo library. The required listener handles the image, whether from the camera or library. Camera is not currently supported on Android devices.

- media.show(*imageSource*, *listener*) –. *imageSource* can be: media.PhotoLibrary, media.Camera, media.SavedPhotosAlbum.

Project 11.1 X-Ray Camera

That’s right, you read the title of this project correctly, and we are going to turn the camera on your smartphone into an X-Ray Camera! This is so much better than those X-Ray glasses that we (okay, I) paid too much for as a child from the back of comic books!

This app definitely falls under the 'joke-app' category. For this app, we will really take someone's picture; do a bit of processing with a mask, then display the skeleton.

Christina Cheek of Art & Design Studios has graciously allowed the use of an illustrated skeleton for our project. You can see more of Christina's work at:
http://artanddesignstudios.hostmyportfolio.com/

Image: Christina Cheek

Before we dive into the code, a couple of notes: This project will currently only work on an iOS device. The camera API is not available through the simulator or on Android. On a Macintosh you are able to select an existing JPG image for testing in the Corona Simulator.

Our config file is standard for most projects.

config.lua

```
application =
{
      content =
      {
            width = 320,
            height = 480,
            scale = "letterbox",
            fps = 30,
            antialias = false,
            xalign = "center",
            yalign = "center"
      }
}
```

Next, we will setup the build settings. My target device is an iPhone, but I have included icon settings for iPhone 4 & iPad.

build.settings

```
-- build.settings for project: Ch11 X  Ray Camera
settings =
{
      androidPermissions =
      {
            "android.permission.CAMERA"
      },
```

```
      iphone =
      {
            plist =
            {
               CFBundleIconFile = "Icon.png",
               CFBundleIconFiles = {
                  "Icon.png",
                  "Icon@2x.png",
               },
            },
      },
      orientation =
      {
            default ="portrait",
            content = "portrait",
            supported =
            {
                  "portrait"
            },
      },
}
```

On to our main file. We will take advantage of the ui.lua, so we will need to start by adding the require command. Next, we will hide the status bar and check to see if the app is running on a device that supports this operation (at this time, an iPhone or Macintosh running the Corona simulator). If the device is not supported, display an appropriate message.

main.lua

```
local ui = require ("ui")

display.setStatusBar(display.HiddenStatusBar)

-- Camera not supported on Android devices in this build.
local isAndroid = "Android" == system.getInfo("platformName")
local isXcodeSimulator = "iPhone Simulator" == system.getInfo("model")

if(isAndroid or isXcodeSimulator) then
       local alert = native.showAlert( "Information", "Camera API not
available on Android or iOS Simulator.", { "OK"})
end
--
```

Load a background and set the color to red. Display a text message to tap the screen to begin.

```
local bkgd = display.newRect( 0, 0, display.contentWidth,
display.contentHeight )
bkgd:setFillColor( 128, 0, 0 )
```

```
local text = display.newText( "Tap anywhere to launch Camera", 0, 0, 
nil, 16 )
text:setTextColor( 255, 255, 255 )
text.x = 0.5 * display.contentWidth
text.y = 0.5 * display.contentHeight
```

Next, create the ‘processing’ function that will be called from a button press event. The processing function will hide the process button, load a scan bar, and transition the scan bar from the top of the screen to the bottom over the course of 2 seconds.

```
local processing = function (event)
	proceedButton.alpha = 0
	local scanbar = display.newImageRect("scan.png", 320, 50)
	scanbar.x = display.contentWidth/2
	scanbar.y=0
	transition.to(scanbar, {y=display.contentHeight, time= 2000})
```

As the scan bar slides down the screen, we face out the photo image and fade in the skeleton.

```
	local skeleton = display.newImageRect ("invertskele.png", 302, 
480)
	skeleton.alpha = 0
	skeleton.x = display.contentWidth/2
	skeleton.y = display.contentHeight/2
	transition.to(image,{alpha = 0, time = 4000}))
	transition.to(skeleton, {alpha=1, time = 5000})
end
```

The sessionComplete function handles the display of the image. After assigning the results from the event (handled after this function), print is used to pass basic information to the terminal to help with troubleshooting. If an image was loaded, it is centered. Finally, this function calls the function above, processing, from a button that is displayed over the top of the image.

```
local sessionComplete = function(event)
	image = event.target

	print( "Camera ", ( image and "returned an image" ) or "session 
was cancelled" )
	print( "event name: " .. event.name )
	print( "target: " .. tostring( image ) )

	if image then
		-- center image on screen
		image.x = display.contentWidth/2
		image.y = display.contentHeight/2
		local w = image.width
		local h = image.height
	print( "w,h = ".. w ..",".. h )
```

```
    proceedButton = ui.newButton{
           default = "button.png",
           text = "Process Image",
           onPress = processing,
           textColor ={255,0,0,255},
           size = 12}
           proceedButton.x = 160
           proceedButton.y = 340
           bkgd:setFillColor(0,0,0)
           bkgd:removeSelf()
     text:removeSelf()
   end
end
```

And here is where the magic happens: media.show calls the camera, and then passes the resulting image to sessionComplete when the user taps the opening screen. Finally, an event listener is used for the background (bkgd) image tap that calls the camera routine.

```
local listener = function( event )
     media.show( media.Camera, sessionComplete )
     return true
end
bkgd:addEventListener( "tap", listener )
```

Note that we haven't saved the image. It is only maintained in memory until the app is closed. We will discuss saving information to the local device in the next chapter.

Summary

As you can see, the media capabilities of Corona are quickly evolving and improving. In this chapter we reviewed using the audio API, and the media API. With the media API we are able to record audio, take pictures, and show movies. Expect to see more great capacities added to Corona in the near future that will augment these capabilities!

Assignments

1) Add a restart button so that the x-ray app does not need to be restarted every time.
2) Add additional graphics to be seen once the image is 'processed'.
3) Create your own mp4 or mov player using the media.playVideo API.
4) Modify the audio player by adding fade in/out controls.

5) Add audio to the x-ray app so that you can add a short message about the photo that was taken.

Chapter 12:
File Storage & SQLite

In this chapter we will examine several ways to read and save data to a mobile device. The ability to access external information that is located on your device is critical to many types of data-intensive applications. For the sake of simplicity, we shall keep our focus limited to files that are already located on the device. To that end, we will examine:

- File location considerations
- Reading from a file
- Writing to a file
- XML/JSON
- Reading from a SQLite database
- Writing to a SQLite database

File IO Considerations

In chapter 11 we briefly discussed possible file locations. There are three file locations currently available to app developers through Corona:

- system.DocumentsDirectory – should be used for files that need to persist between sessions. When used in the simulator, the user's documents directory is used. You can read and write to this directory.
- system.ResourceDirectory – is the folder or directory where your assets are stored. Never change anything in this folder while the app is running. It could invalidate the app and the OS will consider the app malware and refuse to launch. system.ResourceDirectory is assumed (default) when loading assets for your app.
- system.TemporaryDirectory – Just as the name says, is temporary. Only use for in-app data. There is no guarantee that the file will be there the next time the app is used. You can read and write to this directory, just don't expect files to persist between sessions.

Generally, you will only use the resource directory for app specific information that never changes. The documents directory will be used for all types of files that must be updated and persistent between sessions. The temporary directory is those files that are transitory and will not be needed once the app has been closed.

You should think of your apps as being "sandboxed" on any device that they are installed. That means that your files (all of them: images, data, sounds, etc) are stored in a location that is offlimits to any other application that is installed. All of your files will be located in a specific directory for your app.

Reading Data

Implicit vs. Explicit File Manipulation

Lua (and by extension, Corona) has two different types of file manipulation: implicit and explicit.
Implicit file operations use standard, predefined files for file input and output. By default this is the Corona Terminal in Corona, but in code is stdin (standard in), stdout (standard out) and stderr (error reporting).
Explicit file operations allow the reading and writing of typical (i.e. not Corona Terminal) files including text files and binary files. The API libraries are differentiated for the two types of file manipulation. The io API is for implicit, and the file API is for explicit.

Implicit Read

io.type(*filehande*) – checks whether the file handle is valid. Returns the string "file" if the file is open, "closed file" if the file is closed (not in use), and nil if the object is not a file handle.
io.open(*filename_path [, mode]*) – opens a file for reading or writing in string (default) or binary mode. Will create the file if it doesn't already exist. Modes: "r" – read; "w" – write; "a" – append; "r+" – update, all previous data preserved; "w+" – update, all previous data erased; "a+" – append update, all previous data preserved, writing allowed at the end of file. Mode string can include "b" for binary mode.
io.input(*[file]*) – sets the standard input file (default is Corona Terminal)
io.lines(*filename*) – opens the given file in read mode. Returns an iterator (counter) that each time it is called, returns a new line from the file.
io.read(*[fmt]*) – reads the file set by io.input based upon the read format. Generally used with Corona Terminal. Use file:read to for files.
io.close() – closes the open file.
io.tmpfile() – creates an empty, temporay file for reading and writing.

Explicit Read

file:read(*[fmt1] [,fmt2] [, …]*) – reads a file according to the given format. Available formats include: "*n" – reads a number; "*a" – reads the whole file starting at the current position; "*l" – reads the next line (default); number – reads a string with up to the number of characters.
file:lines() – iterates through the file, returning a new line each time it is called.
file:seek(*[mode] [, offset]*) – sets and gets the file position, measured from the beginning of the file. Can be used to get the current file position or set the file position.
file:close() – Close the open file.

Writing Data

Implicit

io.output(*[file]*) – sets the standard output file (default is Corona Terminal).
io.write(*arg1 [, arg2] [, ...]*) – writes the argument to the file. The arguments must be a string or number.
io.flush() – forces the write of any pending io.write commands to the io.output file.

Explicit

file:setvbuf(*mode [, size]*) – set the buffering mode for file writes. Available modes include: "no" – no buffering (can affect app performance); "full" – output only performed when buffer is full or flush; "line" – buffering occurs until a newline is output. Size argument is in bytes.
file:write(*arg1 [, arg2] [, ...]*) – writes the value of each argument to the file. Arguments must be strings or numbers.
file:flush() – forces the write of any pending file:write commands to the file.

JSON

JavaScript Object Notation is a popular lightweight alternative to XML and is fully integrated and supported by Corona. Basic JSON commands include decode, encode, and null. JSON is an external library, so it does need to be loaded with a require "json" prior to use.

json.decode(*json_string*) – decodes the JSON encoded data structure and returns it as a Lua table object.
json.encode(*json_table*) – encodes and returns the lua object as a JSON encoded string.
json.null() – returns a null (decoded as a nil in Lua).

SQLite

The need to be able to access and modify data in any kind of application is critical. As databases are the primary means of storing large quantities of data, any software that is lacking this critical component will quickly be found wanting. As far as developer skills, being able to use and work with databases effectively inside an app is generally considered what separates the intermediate developer from an advanced/skilled developer. Fortunately, Corona has this capability, and soon you can join the ranks of skilled developers!

Corona includes native SQLite support for iOS and compiled version (adding a mere 300K to your app size) for Android. The API for SQLite in Lua is provided by luasqlite3 v0.7. You can find the full documentation on luasqlite3 at http://luasqlite.luaforge.net/lsqlite3.html. Additional information on SQLite can be found at http://www.sqlite.org/lang.html.

While teaching the SQL language is far beyond the scope of this book, there are many great resources. We will look at what is required to create, read, write, and append a SQL file in this chapter.

LuaSQLite Commands

Luasqlite3 is an external library, thus requires the use of

```
require "sqlite3"
```

prior to any SQLite calls.

sqlite3.open(*path)* – opens the SQLite file. Note that the path should be the full path to the database, not just the file name to avoid errors.
sqlite3.version() – returns the version of SQLite in use.
file:exec (*SQL_Command)* – executes a SQL command in the database. Typically used to create tables, insert, update, append or retrieve data from a database.
file:nrows(*SQL_Command)* – returns successive rows from the SQL statement.
file:close() – close the database.

While there are many (many) more SQLite commands available to us, these five basic commands will allow us to get started in developing database-centric apps.

Project 12: Reading a SQLite Database

For this first database project, we will load a SQLite database with zip code data. This project includes data created by MaxMind, available from http://www.maxmind.com/. I used the SQLite Manager plugin for Firefox to manage the import and cleaning up the data for our needs.
The code is fairly straight forward. We will use a standard config.lua and build.settings file:

build.settings

```
settings =
{
	orientation =
	{
		default ="portrait",
		supported =
		{
			"portrait"
		},
	},
}
```

config.lua

```
application =
```

```
{
      content =
      {
            width = 320,
            height = 480,
            scale = "letterbox",
            fps = 30,
            antialias = false,
            xalign = "center",
            yalign = "center"
      }
}
```

First, we will need to import the sqlite3 framework and set the path to your database file and open the associated file.

In this example, the database (zip.sqlite) is located in the same folder as my main.lua file to begin with. That means it should only be read from and not updated or written to in any fashion. This can create problems on some devices as the SQLite trys to keep track of records in the database. To resolve his problem, we will check to see if a copy of the zip.sqlite database is in the app document folder. If it isn't, we will copy it using io read from the resource folder to write the database to the document folder before opening it.

main.lua

```
--include sqlite
 require "sqlite3"

-- Does the database exist in the documents directory
--(allows updating and persistance)
local path = system.pathForFile("zip.sqlite",
system.DocumentsDirectory )
file = io.open( path, "r" )
   if( file == nil )then
      -- Doesn't already exist, so copy it from the
       --resource directory
      pathSource = system.pathForFile( "zip.sqlite",
system.ResourceDirectory )
      fileSource = io.open( pathSource, "r" )
      contentsSource = fileSource:read( "*a" )
      --Write Destination File in Documents Directory
      pathDest = system.pathForFile( "zip.sqlite",
system.DocumentsDirectory )
      fileDest = io.open( pathDest, "w" )
      fileDest:write( contentsSource )
      -- Done
      io.close( fileSource )
      io.close( fileDest )
   end

-- One way or another the database exists
-- So open database connection
```

```
db = sqlite3.open( path )
```

Next, I setup some code to handle an applicationExit event, so that the database will be properly closed should the user hit the home button.

```
-- handle the applicationExit event to close the db
 local function onSystemEvent( event )
   if( event.type == "applicationExit ") then
     db:close()
   end
 end
```

I've included a couple of print statements to show the current version and path being used to help with troubleshooting:

```
print ("version "..sqlite3.version())
print ("db path "..path)
```

Next, I use a select statement, limiting the rows found to 20, and display the content of those rows to my device:

```
local count =0
local sql = "SELECT * FROM zipcode LIMIT 20 "
for row in db:nrows(sql) do
   count = count +1
   local text = row.city.. ", "..row.state.. " "..row.zip
   local t = display.newText(text, 20, 30 +(20 * count),
```

```
native.systemFont, 14)
   t:setTextColor(255,255,255)
end
```

and finally, I setup the system event listener for the close event that was handled earlier.

```
-- system listener for applicationExit
 Runtime:addEventListener (“system”, onSystemEvent)
```

Project 12.1 Writing to a SQLite Database

In this project we are going to do one of the most requested types of apps. A simple form that will be saved to a SQLite database. For the sake of simplicity, I am going to limit the database to a single table. We will make use of textfields (previously discussed in chapter 4) so that real data can be entered. As I spend most of my waking hours working with students, I am going to make this app a simple list of student information. Obviously it could be adapted to any number of different forms or situations.

I am building this app for a tablet device (specifically the iPad) so that I have a little more room for data entry. This app will have three screens: A data entry screen, a list of students in the class, and a beginning screen that will allow the user to select between the two other screens.

To simplify switching between multiple views, we will use director by Ricardo Rauber Pereira. Director is a free download from the Corona site, and version 1.4 is included in the sample files. Director takes advantage of grouping display objects to allow moving between screen views easily. Director, as you may have guessed, can also be used to easily create splash screens for apps.

To simplfy the structure of the finished app, each screen will be stored in its own lua file. This will give us a total of four lua files beyond config.lua: main.lua, menu.lua, addStudent.lua, and displayClass.lua.

First, our build.settings and config.lua files:

build.settings

```
settings =
{
      orientation =
      {
            default ="portrait",
            supported =
            {
                  "portrait","portraitUpsideDown"
            },
```

```
      },

}
```

config.lua

```
application =
{
      content =
      {
            width = 768,
            height = 1024,
            scale = "letterbox",
            antialias = false,
            xalign = "center",
            yalign = "center"
      }
}
```

main.lua

When using Director, the main lua file primarily serves as as starting place for your app. We begin by loading the external file director and creating a new display group.

```
local director = require("director")

-- CREATE A MAIN GROUP
local mainGroup = display.newGroup()
```

Next, we will create a main function that adds the director view to the display group that was just created. Once the directo view controls are added, we can change to the menu screen that is actually menu.lua. The last portion of the main.lua file returns a true value for the function so that the function works correctly.

```
-- MAIN FUNCTION
local main = function ()

      -- Add the group from director class
      mainGroup:insert(director.directorView)

      -- Change scene without effects
      director:changeScene("menu")

      -- Return
      return true
end

main()
```

The final step in our main.lua code is to call the main function that we just created so that director is initiated and control is passed to the menu.lua file. Notice that nothing was

actually displayed in our main.lua file. All we did was initialize director and pass control to menu.lua.

menu.lua

The menu.lua file controls the flow between the different pages of our app. It will display two buttons, one that will load the add student screen and one to display the class roster that is stored in the SQLite database. The first portion of the menu.lua file includes the module(…, package.seeall) command that is required of external files. Then includes comments to detail the contents of the file.

```
module(..., package.seeall)

-- SCENE: Menu

--[[
*****************
 - INFORMATION
*****************
  - Menu.
 Show buttons for selecting the display class screen or add student
screen
 --]]
```

Next we create a function to handle everything for this screen. The new = function (params) is used by director to pass control to and from the menu.lua file. After we create this new function, we load the external file ui.lua to assist with creating our buttons.

```
new = function ( params )

      local ui = require ( "ui" )
```

We will create a local display group for the content for this screen, create two functions to handle the two buttons. Each button will send the user to the respective lua file. Since we

are using director, we can take advantage of some special effects. I've set it to slide to the left or right depending on which button is selected.

```
	local localGroup = display.newGroup()

	local displayClass_function = function ( event )
		if event.phase == "release" then
			director:changeScene( "displayClass", "moveFromLeft" )
		end
	end

		local addStudent_function = function ( event )
		if event.phase == "release" then
			director:changeScene( "addStudent", "moveFromRight" )
		end
	end
```

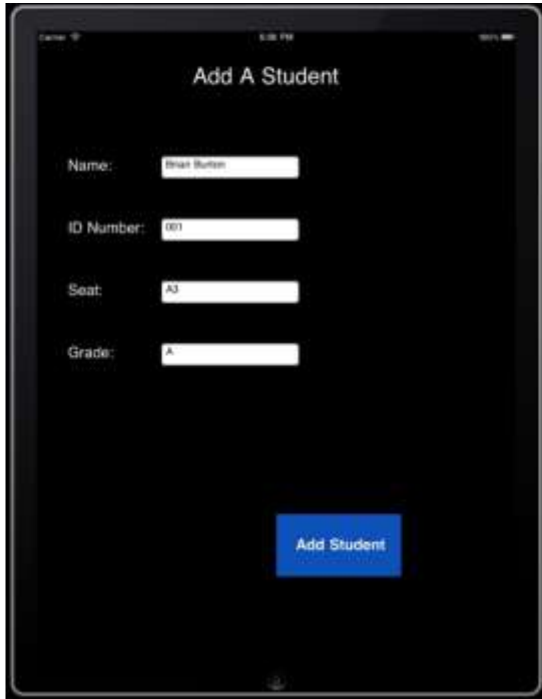
Using the ui.lua external resources, we will create the two buttons and set their respective locations on the iPad.

```
	local  displayClass_button = ui.newButton{
		default = "menuButton.png",
		over = "menuButton.png",
		text = " Class List",
		size = 24,
		emboss=true,
		onEvent = displayClass_function,
		id = "displayClass"
	}

	local addStudent_button = ui.newButton{
		default = "menuButton.png",
		over = "menuButton.png",
		text = "Add a Student",
		size = 24,
		emboss=true,
		onEvent = addStudent_function,
		id = "addStudent"
	}

	addStudent_button.x = display.contentWidth/2
	addStudent_button.y = display.contentHeight/2 - 200
	displayClass_button.x = display.contentWidth/2
	displayClass_button.y=display.contentHeight/2 + 200
```

Finally, we add the two buttons to the display group and return the local group as a parameter for the close of the function.

```
	localGroup:insert(displayClass_button)
	localGroup:insert(addStudent_button)

	return localGroup
```

```
end
```



addStudent.lua

The addStudent.lua module begins with the required module statement and then gives a few comments to describe the function of the file. Creating comments at the beginning of each file is good programming practice. There is nothing worse than trying to figure your program logic 6 months later when the program needs revised or updated.

```
module(..., package.seeall)

--=======================================--
-- SCENE: addStudent
--========================================--

--[[
 - INFORMATION
  - add a student to the SQLite database.
--]]
```

We first create a function called new to hold all of the actions of the addStudent.lua file. This is required by director to easily manage the shift between views. After creating the new function, the external files ui and sqlite are loaded with require; the ui being loaded in a local variable for handling buttons, sqlite being loaded as a global to handle SQLite operations. Finally in this section we create a local group to handle everything that will be

displayed to the screen (required by director to simplify screen management) and local variables are created to store the data we will be gathering for each student.

```
new = function ( params )

  local ui = require ( "ui" )
--include SQLite
  require "sqlite3"

  local localGroup = display.newGroup()

  local studentName
  local studentID
  local seat
  local studentGrade
```

The variables that were just created (studentName, studentID, seat, and studentGrade) must be declared before this next section. Since (as you will see in a few moments) we make reference to the variables in the write (or insert) action to the SQLite database, they must be declared prior to the first SQL write/insert statement, or an error will be generated.

All of the actions associated with writing to the database are contained in one function which is called by the submitStudent button. Once the button is clicked, the path is set to students.sqlite which is stored in the devices documents directory. If the file does not already exist, it is created on the first call. By storing the database in the the documents directory, it will continue to exist from session to session on the device. The statement to open the database is sqlite3.open(path)

```
  -- Setup function for button to write student data
  local submitStudent_function = function ( event )
     if event.phase == "release" then

        -- open SQLite database, if it doesn't exist, create database
        local path = system.pathForFile("students.sqlite",
system.DocumentsDirectory)
        db = sqlite3.open( path )
     -- print(path)   -- for troubleshooting
```

Once the file is open, we need to setup the table that will hold the data. If the table doesn't exist, it will be created. You will notice that we are using standard SQL to define the table (which is called myclass in the example below). The command db:exec(tablesetup) is used to execute the SQL command.

```
        -- setup the table if it doesn't exist
        local tablesetup = "CREATE TABLE IF NOT EXISTS myclass (id
INTEGER PRIMARY KEY, FullName, SID, ClassSeat, Grade);"
        db:exec( tablesetup )
     -- print(tablesetup) -- for troubleshooting
```

After the table is configured, we execute the SQL insert command to pass the contents of the previously created variables to the database. Please note that getting the single quote and double quotes in the right order is critical and is usually the cause of errors in data not being written to the database. The double quotes is used for encapsulating the SQL statement, the single quotes are actually a part of the the SQL statement, as the strings (which all four variables are) must be enclosed in single quotes to pass correctly. The final insert statement after all of the concationation and quotes would read: "INSERT INTO myclass VALUES(NULL,'Brian Burton', '0001', 'A5', 'A');" assuming that I was student 0001, sitting in seat A5 and was earning an A in the class.

```
        -- write student data to database
        local tablefill ="INSERT INTO myclass VALUES (NULL,'" ..
studentName.text .. "','" .. studentID.text .. "','" .. seat.text ..
"','" .. studentGrade.text .."');"
        -- print(tablefill)  -- for troubleshooting
        db:exec( tablefill )
```

After we have executed the SQL insert statement, the database is closed and the app returns control back to the menu.lua file through director.

```
        -- close database
        db:close()
        print("db closed")

       -- return to menu screen
       director:changeScene( "menu", "moveFromLeft" )
    end  --if statement
  end   -- submitStudent_function
```

Next we create the addStudent button using the external ui routines.

```
  local  addStudent_button = ui.newButton{
     default = "selectButton.png",
     over = "selectButton.png",
     text = " Add Student",
     size = 24,
     emboss=true,
     onEvent = submitStudent_function,
     id = "addStudent"
  }
  addStudent_button.x = display.contentWidth/2+100
  addStudent_button.y = display.contentHeight-200
```

After adding the button to the screen, labels and textfields are added to the screen to capture the input data. Each of the labels is added to the local display group at the time of creation. Remember that textfields do not currently display in the Corona simulator. I recommend that you use dummy data in place of the textfields initially.

```
  -- Add textboxes to enter data
     -- Labels for textboxes
```

```
  local title = display.newText(localGroup, "Add A Student", 250, 50,
native.systemFont, 36)
  title:setTextColor(255,255,255)

  local nameLabel = display.newText(localGroup, "Name:", 50, 200,
native.systemFont, 24)
  nameLabel:setTextColor(255,255,255)

  local idLabel = display.newText(localGroup, "ID Number:", 50, 300,
native.systemFont, 24)
  idLabel:setTextColor(255,255,255)

  local seatLabel = display.newText(localGroup, "Seat:", 50, 400,
native.systemFont, 24)
  seatLabel:setTextColor(255,255,255)

  local gradeLabel = display.newText(localGroup, "Grade:", 50, 500,
native.systemFont, 24)

  studentName = native.newTextField(200, 200, 220, 36)
  studentName.inputType="default"

  studentID = native.newTextField(200, 300, 220, 36)
  studentID.inputType="default"

  seat = native.newTextField(200, 400, 220, 36)
  seat.inputType="default"

  studentGrade = native.newTextField(200, 500, 220, 36)
  studentGrade.inputType="default"

  -- add all display items to the local group
  localGroup:insert(addStudent_button)
```

Remember to include a routine to properly close the database should the app unexpectedly close.

```
  -- handle the applicationExit event to close the db
  local function onSystemEvent( event )
     if( event.type == "applicationExit") then
        db:close()
      end
  end

  -- system listener for applicationExit to handle closing database
  Runtime:addEventListener ("system", onSystemEvent)

  return localGroup
end
```

displayClass.lua

The displayClass.lua file handles all of the displaying of all student data to the device screen. As usual, we start with the module command and describing the function of this file. Followed by creating the required function for director and loading the required external files.

```
module(..., package.seeall)

--============================================--
-- SCENE: displayStudent
--============================================--

--[[
*****************
 - INFORMATION
*****************
  - Display class information to screen.
--]]

new = function ( params )

  local ui = require ( "ui" )
  --include SQLite
  require "sqlite3"

  local localGroup = display.newGroup()
```

Next we will open the database.

```
  -- open database
  local path = system.pathForFile("students.sqlite",
system.DocumentsDirectory)

  db = sqlite3.open( path )
  --  print(path)
```

Now we will create the SQL statement that will return all of the fields from the database. We will use the for row in db:nrows(sql) command so that we can step one row at a time through the data.
In this case, row is being used as a variable to hold the data that is returned from the db:nrows(sql) command. This allows us to loop through the result set that is returned by the SQL statement and work with each row (or set of data). As you can see, we then take the fields from the row and create a text object that is then displayed to the device screen.

```
  --print all the table contents
  local sql = "SELECT * FROM myclass"
  for row in db:nrows(sql) do
     local text = row.FullName.." "..row.SID..", "..row.ClassSeat.."
"..row.Grade
     local t = display.newText(text, 20, 30 * row.id,
native.systemFont, 24)
     t:setTextColor(255,255,255)
  end

  db:close()  -- finished with the database, so close it.
```

Now just a little house keeping: we create the function and button to handle returning to the menu; add a routine to handle unexpected app closing to close the database properly; and finally return to the director call.

```
  -- Setup function for button to load student data
  local displayClass_function = function( event )
     if event.phase == "release" then
          -- return to menu screen
          director:changeScene( "menu", "moveFromRight" )
     end
  end

  local  displayClass_button = ui.newButton{
     default = "selectButton.png",
     over = "selectButton.png",
     text = " Return to Menu",
     size = 24,
     emboss=true,
     onEvent = displayClass_function,
     id = "displayClass"
  }
  displayClass_button.x = display.contentWidth/2+100
```

```
  displayClass_button.y = display.contentHeight-200

  -- add all display items to the local group
  localGroup:insert(displayClass_button)


  -- handle the applicationExit event to close the db
  local function onSystemEvent( event )
     if( event.type == "applicationExit") then
          db:close()
     end
  end

  -- system listener for applicationExit to handle closing database
  Runtime:addEventListener ("system", onSystemEvent)

  return localGroup

end
```

Summary

While working with databases and external files can be challenging in the beginning, the functionality and data storage efficiency that they provide cannot be beat. In this chapter we have examined how to use external files and databases.

Assignments

1) Modify Project 12.1 to include the student's gender and age.

2) Augment Project 12.1 to include the ability to update student information (advanced project).

3) Create a database app to store the current date and temperature. The retrieve page should list the dates and temperature.

4) Create a highschore app that accepts the name of the game, name of player, high score, and the date the high score was achieved.

Chapter 13: Waiting on Tables

In this chapter we are going to begin working with Tables. Tables have become a critical part of mobile application development, with Apple having spent enormous amount of effort in the creation and refinement of tables for the iPhone and iPad. Tables are one of the simplest ways to display large quantities of data that eventually provide detailed information.

In our examination of tables we will:

- Clarify the term table
- Examine the tools available for tables
- Create a simple table
- Create a complex table, loading from a SQLite database

Table vs. Table: Clearing up the Confusion

The term table has many different meanings in programming. It can be used to refer to an array (which is the common usage in Lua), a grid layout, or a table view (popularized by Apple for developing data intensive applications) sometimes also referred to as a list view. For the purposes of this chapter (and all chapters in this book), I will only use the term table to refer to the table view associated with app development that has been used by Apple. If I am referring to an array table (a term commonly used in Lua), I will specify it as an array or a Lua array table.

I should note that Corona does have a table command in the API. This refers to the Lua array table. On the Ansca Mobile website you will see the concept of table views referred to as table views and list views.

Tools for Tables

We have two sets of tools available for creating table views: the widget and an external library. The widget at the time of this writing is only available for iOS devices, and is addressed in chapter 15. The external library works on any device and will be the focus of this chapter.

The external library was created by Gilbert Guerrero. He has been updating the library on a regular basis. While I have included the external library in the project files, you can download the most recent version of the library from http://bitbucket.org/gilbert/table-view-library/src. At the time of this writing there are two versions of the library; the original, which allows grouping and scrollTo functions, and the XL version, which is faster,

but doesn't include all of the features. As the database I will be using is relatively small, I have elected to use the original version for the projects in this chapter.

By adding the table view external file, we gain the following arguments:

- data – an array containing elements that the list can iterate through
- default – a background image for the row. Used to define the hit area for touches
- over – image shown on touch
- background color – provided in the standard R, G, B format
- callback – a function to define how the data is displayed in each row
- onRelease (optional)- function to call after tap
- top – upper boundary of the list
- bottom – bottom boundary of the list
- cat- array key name used to hold category values for each item (used for creating groups)
- order (optional) – allows for specifying an order for headers

and a few methods are provided as well:

- myList:addScrollBar() – adds a scroll bar to the screen
- myList:removeScrollBar() – removes the scroll bar from the screen
- myList:addOnTop(object, x, y) – appends an object to the top of the list. Can be used to add a search bar
- myList:addOnBottom(object, x, y) – appends an object to the end of the list
- myList:scrollTo(y, time) – automatically scrolls the list to a location (such as a return to top)
- myList:cleanUp() – destroys the list, clears reserved memory and stops the event listeners.

Project 13: Creating a Simple Table View

For this first project, we are going to create a simple table view that lists places you have lived or traveled. We will use the tableview library for this project and all the data will be contained within the program. I am specifically targeting phones for this project; either the iPhone or a Droid.

build.settings

```
settings =
{
      orientation =
      {
            default ="portrait",
            supported =
            {
                  "portrait"
```

```
            },
      },
}
```

config.lua

```
application =
{
      content =
      {
            width = 320,
            height = 480,
            scale = "letterbox",
            fps = 30,
            antialias = false,
            xalign = "center",
            yalign = "center"
      }
}
```

Within the main.lua file, we will begin by hiding the status bar and loading the ui and table view external libraries. Then we will declare a few variables that will be used later in our program.

main.lua

```
display.setStatusBar( display.HiddenStatusBar )
```

```
local ui = require("ui")
local tableView = require("tableView")

local myList, backBtn, detailScreenText
local screenOffsetW, screenOffsetH = display.contentWidth -
display.viewableContentWidth, display.contentHeight -
display.viewableContentHeight
```

Now let's create some data. I am using a two-dimensional array to store the data.

```
-- load the array to be displayed in the table view
local data = {}
-- set each row of the array as an array, then load state name and
abbreviation
data[1] = {}
data[1].state ="California"
data[1].abbrev="CA"

data[2] = {}
data[2].state ="Indiana"
data[2].abbrev="IN"

data[3] = {}
data[3].state ="Massachusetts"
data[3].abbrev="MA"

data[4] = {}
data[4].state ="Missouri"
data[4].abbrev="MO"

data[5] = {}
data[5].state ="Texas"
data[5].abbrev="TX"
```

Using display groups, we will create a detail display that will show which item was tapped.

```
--setup a destination for the list items
local detailScreen = display.newGroup()

local detailBg = display.newRect(0,0,display.contentWidth,
display.contentHeight-display.screenOriginY)
detailBg:setFillColor(255,255,255)
detailScreen:insert(detailBg)

detailScreenText = display.newText("You tapped item", 0, 0,
native.systemFontBold, 16)
detailScreenText:setTextColor(0, 0, 0)
detailScreen:insert(detailScreenText)
detailScreenText.x = math.floor(display.contentWidth/2)
detailScreenText.y = math.floor(display.contentHeight/2)
detailScreen.x = display.contentWidth
```

Now for a few functions. First, a function to handle the event when someone taps one of the listed states. The tapped state will be stored in the variable self. The text to be displayed in the detail view is set to a statement giving the state name and abbreviation. Then we will do a few transitions, sliding the list view out and the detail screen into view.

```
--setup functions to execute on touch of the list view items
function listButtonRelease( event )
      self = event.target
      print(self.id)
      detailScreenText.text = "The abbrev. for ".. data[self.id].state
.." is "..data[self.id].abbrev

      transition.to(myList, {time=400, x=display.contentWidth*-1,
transition=easing.outExpo })
      transition.to(detailScreen, {time=400, x=0,
transition=easing.outExpo })
      transition.to(backBtn, {time=400, x=math.floor(backBtn.width/2)
+ screenOffsetW*.5 + 6, transition=easing.outExpo })
      transition.to(backBtn, {time=400, alpha=1 })

      delta, velocity = 0, 0
end
```

Now to handle events should the back button be tapped. This is a simple function that will slide the detail view off screen and the original table view back on to the screen.

```
function backBtnRelease( event )
      print("back button pressed")
      transition.to(myList, {time=400, x=0, transition=easing.outExpo
})
      transition.to(detailScreen, {time=400, x=display.contentWidth,
transition=easing.outExpo })
      transition.to(backBtn, {time=400,
x=math.floor(backBtn.width/2)+backBtn.width, transition=easing.outExpo
})
      transition.to(backBtn, {time=400, alpha=0 })

      delta, velocity = 0, 0
end
```

Time to setup the table view. Using the external table view library, we will pass data (which we called data... I know, real original). Next we set the background image assigned to default, the image to display when the table item is tapped (the over image), and the function to call when the tap is completed (onRelease).
Top passes the distance from the top of the screen for the list to start. Bottom passes the distance from the bottom of the screen for the list to snap back to when scrolled.
The critical element is the callback parameter. Callback details how to display each item in the list. In this example, each item (or state, loaded from the data variable) is assigned as a newText item to the variable t. The color of the text is set to white, and the .x and .y parameters set the x and y location of each element in the table.

```
myList = tableView.newList{
        data=data,
        default="listItemBg.png",
        over="listItemBg_over.png",
        onRelease=listButtonRelease,
        top=60,
        bottom=1,
        callback=function(item)
            local t = display.newText(item.state, 0, 0,
native.systemFontBold, textSize)
            t:setTextColor(255, 255, 255)
            t.x = math.floor(t.width/2) + 20
            t.y = 46
            return t
        end
}
```

Now we are ready to setup the navigation text and navigation bar. We will create the navigation bar as a button so that when it is tapped the scrolling will automatically return to the top of the table view.

```
--Setup the nav bar
local navBar = ui.newButton{
      default = "navBar.png",
      onRelease = scrollToTop
}
navBar.x = display.contentWidth*.5
navBar.y = math.floor(display.screenOriginY + navBar.height*0.5)

local navHeader = display.newText("Places I've Lived", 0, 0,
native.systemFontBold, 16)
navHeader:setTextColor(255, 255, 255)
navHeader.x = display.contentWidth*.5
navHeader.y = navBar.y
```

And finally we create the back button. The back button is centered to the y axis center of the navigation bar and currently set to an alpha of 0 so that it is not visible on the first table view screen. When the state is tapped and the detail view is loaded, the back button alpha will be reset to 1 so that it is visible.

```
--Setup the back button
backBtn = ui.newButton{
      default = "backButton.png",
      over = "backButton_over.png",
      onRelease = backBtnRelease
}
backBtn.x = math.floor(backBtn.width/2) + backBtn.width +
screenOffsetW
backBtn.y = navBar.y
backBtn.alpha = 0
```

That is an example of a simple table view. Now let's try something a little more complex: a 4 screen table view that loads from a SQLite database.

Project 13.1: Table View From SQLite

I tried this project a few different ways before developing the example that is detailed over the next several papges. In the first attempt, I tried to load 43,000 + records into memory, just to see if it could be done. It locked-up the Corona simulator and the iPhone 3G I was testing on. What can we take away from this? Loading all of your data when you don't need to is a bad idea. With that in mind, I set out to develop a project that would show better ways of handling mobile device memory limitations and a way to segment your data for efficiency.

Project 13.1 is designed to allow the user to select a state and city to show what zip codes are used in that city. To improve efficiency, only the state data is loaded initially. After a state is selected, then the corresponding cities are displayed. Finally, after selecting a city, the zip code(s) for the selected state and city are displayed. Back buttons are included through out the program so that the user is able to easily navigate back and forth.

We will be using Gilbert Guerrero's table view external library once again so that the project can be deployed to any iOS or Android device. I have created this app with the iPhone in mind.

build.settings

```
settings =
{
	orientation =
	{
		default ="portrait",
		supported =
		{
			"portrait"
		},
	},
}
```

config.lua

```
application =
{
	content =
	{
		width = 320,
		height = 480,
		scale = "letterbox",
		fps = 30,
		antialias = false,
```

```
            xalign = "center",
            yalign = "center"
      }
}
```

main.lua

We will start our main.lua file off by hiding the status bar and loading the external libraries required for this app.

```
display.setStatusBar( display.HiddenStatusBar )

local ui = require("ui")
local tableView = require("tableView")
require("sqlite3")
```

Now let's setup the variables that will be used in this project. As you can see, the stateData, cityData, and zipData are all created as arrays to store the associated data. We will also store the screen offset for width and height so that the different views can be easily exchanged by the tableView library.

```
local stateList, cityList, zipList, backBtn, detailScreenText,
selectState, selectCity
local stateData={}
local cityData ={}
local zipData={}
local zipScreen = display.newGroup()
local screenOffsetW, screenOffsetH = display.contentWidth -
display.viewableContentWidth, display.contentHeight -
display.viewableContentHeight
```

Since the navBar will be used by all the different pages of the table view, we need to declare it early in the program. The navBar does have the properties of a button and when tapped will start a scroll to top event for the tableView library.

```
--Setup the nav bar
local navBar = ui.newButton{
      default = "navBar.png",
      onRelease = scrollToTop
```

```
}
navBar.x = display.contentWidth*.5
navBar.y = math.floor(display.screenOriginY + navBar.height*0.5)
```

Now let's create three different headers for the navBar, depending on which view is currently visible. Since the app starts with the state view, we will leave the alpha as 1, and set the other nav headers to an alpha of 0.

```
local navStateHeader = display.newText("Select a State", 0, 0,
native.systemFontBold, 16)
navStateHeader:setTextColor(255, 255, 255)
navStateHeader.x = display.contentWidth*.5
navStateHeader.y = navBar.y

local navCityHeader = display.newText("Select a City", 0, 0,
native.systemFontBold, 16)
navCityHeader:setTextColor(255, 255, 255)
navCityHeader.x = display.contentWidth*.5
navCityHeader.y = navBar.y
navCityHeader.alpha = 0

local navZipHeader = display.newText("Zip codes for selected city ",
0, 0, native.systemFontBold, 16)
navZipHeader:setTextColor(255, 255, 255)
navZipHeader.x = display.contentWidth/2+20
navZipHeader.y = navBar.y
navZipHeader.alpha=0
```

Ready to setup our database. First we will see if it already exists in the documents directory. If it doesn't, it will be copied from the resource folder to the document directory. After this check (or copy) we can safely open the database.

```
-- Does the database exist in the documents directory
--(allows updating and persistance)
local path = system.pathForFile("zip.sqlite",
system.DocumentsDirectory )
file = io.open( path, "r" )
   if( file == nil )then
  -- Doesn't already exist, so copy it in from resource
  -- directory
      pathSource     = system.pathForFile( "zip.sqlite",
system.ResourceDirectory )
      fileSource = io.open( pathSource, "r" )
      contentsSource = fileSource:read( "*a" )

      --Write destination file in documents directory
      pathDest = system.pathForFile( "zip.sqlite",
system.DocumentsDirectory )
      fileDest = io.open( pathDest, "w" )
      fileDest:write( contentsSource )
```

```
      -- Done
      io.close( fileSource )
      io.close( fileDest )
   end
-- One way or another the database file now exists
-- So open database connection
db = sqlite3.open( path )
```

The exit function needs to handle a few more operations now. To begin with, we will close the database, then clear the state, city, and zip arrays to ensure that the program does a clean close.

```
-- handle the applicationExit event to close the db
local function onSystemEvent( event )
      if( event.type == "applicationExit") then
            db:close()
            stateList:cleanUp()
            cityList:cleanUp()
            zipList:cleanUp()
      end
end
```

Detail View (part of the main.lua file)

Time to setup the different views. Since each view calls the next view sequentially, the last view must be listed in code first.

The first view to be listed in our code will be the detail view which is the result of the user tapping or selecting a zip code from the zip code view. I have named the function selectZipButtonRelease – i.e. when the user selects a zip code, on the release of the tap, this event is called.

The detail view is a created as a display group, which is named detailScreen. We will start by creating the new group, setup a background (called detailBg) which is a white rectangle the size of the device display.

```
function selectZipButtonRelease(event)
    --setup a destination for the detail item
    local detailScreen = display.newGroup()

    local detailBg = display.newRect(0,0, display.contentWidth,
display.contentHeight-display.screenOriginY)
    detailBg:setFillColor(255,255,255)
    detailScreen:insert(detailBg)
```

As the detail view is the result of an event from the zip code view, we need to know which zip code was tapped. We can capture this information using the event.target data and storing it in a local variable called self. This allows us to show the zipData that was tapped and build some text to be displayed in the view.

```
    local self = event.target
    local tempText = zipData[self.id].." is the zip code for
"..selectCity..", "..selectState
    detailScreenText = display.newText(tempText, 0, 0,
native.systemFontBold, 14)
    detailScreenText:setTextColor(0, 0, 0)
    detailScreen:insert(detailScreenText)
    detailScreenText.x = math.floor(display.contentWidth/2)
    detailScreenText.y = 10
    detailScreen.x = display.contentWidth
```

The final part of the selectZipButtonRelease function is to transition the from previous view (zipScreen) to the new view, detailScreen.

```
    transition.to(zipScreen, {time=400, x=display.contentWidth*-1,
transition=easing.outExpo})
    transition.to(detailScreen, {time=400, x=0,
transition=easing.outExpo})
end
```

Please note that the external library automatically handles a tap event in the detail view to return to the previous view, zipScreen, for which we will now write the code.

ZipScreen view (part of main.lua)

The selectCityButtonRelease is called when the user selects a city in the cityList. The selectCityButtonRelease function generates zipScreen for displaying the zip code results. As in the previous method, we first use the local variable to capture the which city was tapped and save that information to selectCity (which was declared as a local variable at the beginning of our app).
Next, the sql statement is set, specifying the state and city that were selected by the user. The SQL statement will be used in the db:nrows call to return an array of zip codes associated with the selected city and state. These zip codes will be stored in zipData. Please note the structure of the SQL statement, specifically the double quote and single quote marks. A single quote is required around the state and city names since they are strings being passed to the database for comparison. The final output of the SQL statement will look something like this:

```
SELECT zip FROM zipcode WHERE state = 'AL' AND city = 'Abernant'
```

If you are having problems figuring out where to place the single quotation mark, remember that they are a part of the larger string to be quoted; they are a part of what is being quoted by your double quotation mark.

```
-- after the State and City are selected, show available zip codes
function selectCityButtonRelease(event)
      local self = event.target
      selectCity = cityData[self.id]
      local count = 0
      local sql = "SELECT zip FROM zipcode WHERE state = 
'"..selectState.."' AND city = '"..selectCity.."'"
      --print(sql)
      for row in db:nrows(sql) do
            count = count +1
            zipData[count]=row.zip
      end
```

After generating our row data, we will set the zipScreen's alpha to 1 (since it might have been hidden previously). We will also create a white background for the zipScreen.

```
    zipScreen.alpha = 1
    local detailBg = display.newRect(0,0, display.contentWidth,
display.contentHeight-display.screenOriginY)
    detailBg:setFillColor(255,255,255)
    zipScreen:insert(detailBg)
```

The variable zipList will hold the data returned from the call to tableView, which builds the table for our app. We will pass the required parameters (detailed earlier in the chapter). For simplicity, I added each of the items to zipScreen so that we can easily hide or show the content.

```
 zipList = tableView.newList{
    data=zipData,
    default="listItemBg.png",
    over="listItemBg_over.png",
    onRelease=selectZipButtonRelease,
    top=60,
    bottom=1,
    callback=
        function(item)
            local t = display.newText(zipScreen, item, 20, 0,
native.systemFontBold, textSize)
            t:setTextColor(0,0,0)
            t.x = math.floor(t.width/2) + 20
            t.y = 46
            return t
        end
    }
```

The final portion of the selectCityReleaseButton is to handle the transitions. This transition is a little more complex than the previous set that we handled in selectZipReleaseButton. To begin with, we hide the cityList, and show zipScreen. We are also bringing into view the zipBackBtn and bring to front the navBar, zipBackBtn, and the navZipHeader to ensure that they are not being covered by one of the other views. This is a critical step; remember that everything added to the display is added on top of anything previously displayed. It is like laying multiple sheets of paper on a desk. Each successive layer covers the previous later.

```
      transition.to(CityList, {time=400, x=display.contentWidth*-1,
transition=easing.outExpo })
      transition.to(zipScreen, {time=400, x=0,
transition=easing.outExpo })
      transition.to(zipBackBtn, {time=400,
x=math.floor(backBtn.width/2) + screenOffsetW*.5 + 6,
transition=easing.outExpo, alpha=1 })
      navBar:toFront()
      zipBackBtn:toFront()
      navZipHeader:toFront()
      transition.to(navZipHeader, {time=400,alpha = 1})
      transition.to(navCityHeader, {time=400, alpha = 0})
```

```
      delta, velocity = 0, 0
end
```

cityList view (part of main.lua)

The selectStateButtonRelease calls our previous routine: selectZipButtonRelease. The selectStateButtonRelease function narrows the zip code selection by city. The first few lines are similar to what we used in selectZipButtonRelease: we set a local self variable to the event target, set the selected State for later use and loaded our cityData array with information from the SQLite database.

```
--setup function to execute on touch of the state list/table view
items
function selectStateButtonRelease( event )
      local self = event.target
      selectState = stateData[self.id]
      --print(selectState)

-- pull city data from database for selected state
      local count = 0
      local sql = "SELECT DISTINCT city FROM zipcode WHERE state = '"..
selectState.."'"
      for row in db:nrows(sql) do
            count = count +1
            cityData[count]=row.city
            --print(cityData[count])
      end
```

Now we will set up the cityList usingthe tableView external library, passing the appropriate parameters. After setting up the cityList, all that's left for this routine is to handle the transitions as we did in the previous function.

```
  -- list cities thru tableView
  cityList = tableView.newList{
        data=cityData,
        default="listItemBg.png",
        over="listItemBg_over.png",
        onRelease=selectCityButtonRelease,
        top=60,
        bottom=1,
        callback=function(item)
            local t = display.newText(item, 0, 0,
native.systemFontBold, textSize)
            t:setTextColor(255, 255, 255)
            t.x = math.floor(t.width/2) + 20
            t.y = 46
            return t
        end
}
   -- handle screen transitions
   transition.to(stateList, {time=400, x=display.contentWidth*-1,
transition=easing.outExpo })
   transition.to(cityList, {time=400, x=0, transition=easing.outExpo
})
   transition.to(backBtn, {time=400, x=math.floor(backBtn.width/2) +
screenOffsetW*.5 + 6, transition=easing.outExpo, alpha=1 })
   transition.to(navStateHeader, {time=400, alpha=0})
   transition.to(navCityHeader, {time=400, alpha = 1})

   delta, velocity = 0, 0
end
```

stateList view (part of main.lua)

The getState function is a little simplier than our previous functions since it is the first screen called. We don't have to worry about handling previous screens, just load the list of states (which includes United States territories, in case you were wondering about the AS abbreviation).

```
local getState = function()
  -- load the state array to be displayed in the table view
  -- set each row of the array as an array, then load state name and
abbreviation
  local count =0
  local sql = "SELECT DISTINCT state FROM zipcode"
  for row in db:nrows(sql) do
       count = count +1
       stateData[count] = row.state
       --print(stateData[count])
  end
  stateList = tableView.newList{
        data=stateData,
        default="listItemBg.png",
        over="listItemBg_over.png",
        onRelease=selectStateButtonRelease,
        top=60,
        bottom=1,
        callback=function(item)
           local t = display.newText(item, 0, 0,
native.systemFontBold, textSize)
           t:setTextColor(255, 255, 255)
           t.x = math.floor(t.width/2) + 20
           t.y = 46
           return t
       end
  }
  stateList:addScrollBar()
```

```
end
```

The next function, cityBackBtnRelease, handles the backbutton for the cityList. If the user taps the back button on the cityList, then cityBackBtnRelease reloads the stateList, hides the back button and disposes of the cityList to free memory and prepare for a new city to be selected.

```
function cityBackBtnRelease( event )  -- reload the state table view
      --print("city back button released")
      transition.to(stateList, {time=400, x=0, 
transition=easing.outExpo })
      transition.to(cityList, {time=400, x=display.contentWidth, 
transition=easing.outExpo })
      transition.to(backBtn, {time=400, 
x=math.floor(backBtn.width/2)+backBtn.width, transition=easing.outExpo 
})
      transition.to(backBtn, {time=400, alpha=0 })
      transition.to(navStateHeader, {time = 400, alpha = 1})
      navStateHeader:toFront()
      transition.to(navCityHeader, {time=400, alpha=0})
      cityList:cleanUp()
      delta, velocity = 0, 0
end
```

Just as the last function handled the back button from the cityList, zipBackButtonRelease handles the back button for the zipScreen to reload the list of cities.

```
function zipBackBtnRelease( event )  --reload the City table view
      --print("zip back button released")
      zipScreen.alpha=0
      transition.to(cityList, {time=400, x=0, transition=easing.outExpo 
})
      cityList:toFront()
      transition.to(zipScreen, {time=400, x=display.contentWidth, 
transition=easing.outExpo })
      transition.to(zipBackBtn, {time=400, 
x=math.floor(zipBackBtn.width/2)+zipBackBtn.width, 
transition=easing.outExpo, alpha=0  })
      transition.to(backBtn, {time=400, alpha = 1})
      backBtn:toFront()
      transition.to(navCityHeader, {time=400, alpha=1})
      navCityHeader:toFront()
      transition.to(navZipHeader,{time=400, alpha=0})
      zipList:cleanUp()
      delta, velocity = 0, 0
end
```

Just a few things left to do. We need to setup the back button to call cityBackBtnRelease and zipBackBtn to handle zipBackBtnRelease. Initially we will set both button’s alpha to 0 since they won’t need to be visible until a state is selected.

```
--Setup the city view back button
backBtn = ui.newButton{
	default = "backButton.png",
	over = "backButton_over.png",
	onRelease = cityBackBtnRelease
}
backBtn.x = math.floor(backBtn.width/2) + backBtn.width + 
screenOffsetW
backBtn.y = navBar.y
backBtn.alpha = 0

--Setup the zip view back button
zipBackBtn = ui.newButton{
	default = "backButton.png",
	over = "backButton_over.png",
	onRelease = zipBackBtnRelease
}
zipBackBtn.x = math.floor(zipBackBtn.width/2) + zipBackBtn.width + 
screenOffsetW
zipBackBtn.y = navBar.y
zipBackBtn.alpha = 0
```

Finally, we call getState to get the program running and setup the listener for when the app is closed.

```
getState()

-- system listener for applicationExit
Runtime:addEventListener ("system", onSystemEvent)
```

Summary

In this chapter we have looked at how to develop simple and complex table view apps. Table views are critically important in mobile app development since they allow the parsing and viewing of large chunks of data in a system that can be managed on a mobile device. In chapter 15 we will look at the widget method for using table views.

Assignments

1) Create a simple table app that will display your top 10 favorite foods. When selected, it should display your favorite side foods to have with the selected item.

2) Challenge Project: Create your favorite foods list as above, adding in categories for breakfast, lunch, and dinner. Make sure the app sorts the foods based upon the category.

3) Create an app that displays a table of sports teams by city. For an additional challenge, sort the teams by sport.

4) Create an app that displays a table of your favorite books organized by author. For an additional challenge, sort the books by genre.

5) Create an app that displays a table of your favorite songs organized by artist. In the detail view, include information about the song or the lyrics.

Chapter 14:
It's Who you Know: Networking

How can we discuss mobile app development without addressing the ability to communicate with a network? In this chapter we will examine the basic methods used to create network connectivity and how to connect with some of the most popular services. We shall:

- Create a network connection
- Check network status
- Connect to a webserver
- Download/upload to a webserver
- Connect to Facebook
- Introduce services such as Papaya and OpenFient
- Introduce inMobi functions for in-app ads
- Introduce in-app credits
- Introduce pubnub for multi-player apps

Web Services

Corona manages web services through the LuaSocket libraries. Through the various modules included in these libraries you can manage web access (HTTP), send emails (SMTP), upload and download files (FTP), as well as filter data (using LTN12), manipulate URLs and support MIME.

Network access is a huge topic. I could (and might in the future) write a book that exclusively covered just networking topics (it would put by CCNA teaching certificate to go purpose). For the time being, we are going to keep to the basics of networking and how to implement network features in your apps.

Over the past year a multitude of libraries and external services have become available to Corona. These services are essential to network based games and apps, so I am also going to briefly introduce these services and explain how to implement them in your game or app.

HTTP

We will begin with the basics of network communication; establishing a connection using hyper-text transfer protocol (http). As most people are use to seeing and using http through their web-browser, using it as part of an app should make sense. The features

available with asynchronous http allow you to make regular calls as well as secure socket layer (SSL) calls.

You can use either the built in network library (does not need a require) or you can use the socket library (which does need a require). We will begin with the network library:

- network.request(url, method, listener [, params]) – makes an asynchronous request (either http or https) to an URL. Time-out for a network is 30 seconds.
    - url – the requested URL
    - method – either GET or POST
    - listener – function to handle the call response. Will return either event.response or event.isError
    - params – a table comprised of params.headers and params.body
- network.download(url, method, listener [, params], destFilename [, baseDir]) – much like network.request, except it downloads the response as a file instead of storing it in memory. Great for XML/JSON documents and images.
- display.loadRemoteImage(url, method, listener [, params], destFilename [, baseDir] [, x, y]) – similar to network download, but specifically designed to load the image from the network.

Project 14: Picture Download – Via Network Library

Our first project in this chapter will demonstrate how to download an image from the network using the network library, save the image to the documents directory, and display it to the screen. For this project I have placed an image on my web server to demonstrate the download method.

config.lua

```
application =
{
	content =
	{
		width = 320,
		height = 480,
		scale = "letterbox",
		fps = 30,
		antialias = false,
		xalign = "center",
		yalign = "center"
	}
}
```

build.settings

We are adding a new line to the standard build.settings. To the androidPermissions we need to add permission for internet access.

```
settings =
{
      androidPermissions =
      {
            "android.permission.INTERNET"

      },

      orientation =
      {
            default ="portrait",
            content = "portrait",
            supported =
            {
                  "portrait"
            },
      },
}
```

main.lua

I use one button in this app, so we'll use the ui library. The networkListener function is used to check to see if there were any network errors. If there were no errors, then the image is displayed.

```
local ui = require("ui")

local function networkListener (event)
      if (event.isError) then
            print("Network error - download failed")
      else
            testImage =
display.newImage("MobileAppDevelopmentCover.png",
system.DocumentsDirectory,10,30);
          testImage.x = display.contentWidth/2
      end
 end
```

I used network.download for this demonstration. First turn on the network activity indicator. Then the URL parameter is set to point at the image on the webserver. GET is our http method. The networkListener function is called to handle the image when it is received from the GET call. The final image is saved to the system.DocumentDirectory as MobileAppDevelopmentCover.png.

```
local function loadButtonPress (event)
  native.setActivityIndicator( true )

network.download("http://www.BurtonsMediaGroup.com/MobileAppDevelopmen
tCover.png", "GET", networkListener, "MobileAppDevelopmentCover.png",
system.DocumentsDirectory)
  native.setActivityIndicator( false )

end

-- Load Button
loadButton = ui.newButton{
      default = "buttonBlue.png",
      over = "buttonBlueOver.png",
      onPress = loadButtonPress,
      text = "Load Picture",
      emboss = true
      }
loadButton.x = display.contentWidth/2
loadButton.y = display.contentHeight - 50
```

Socket

If you are looking for more control over the network interaction, you can use the LuaSocket. Full documentation on the LuaSocket is available from http://www.tecgraf.puc-rio.br/~diego/professional/luasocket/reference.html. To demonstrate how LuaSocket works, I have re-worked Project 14 as a LuaSocket project to show the differences.

Project 14a: Picture Download – Via Socket Library

Config.lua and build.settings remain the same for this project. All of the changes occur in main.lua.

main.lua
We will use three requires in this version: ui, socket.http, and ltn12. Socket handles the http request. ltn12 provides the ability to save data to a file (referred to as a 'sink').
After the necessary require statements, we will set the path and create a file to store the image.

```
local ui = require("ui")
```

```
-- Load the relevant LuaSocket modules
local http = require("socket.http")
local ltn12 = require("ltn12")

-- Create local file for saving data
local path = system.pathForFile( "MobileAppDevelopmentCover.png",
system.DocumentsDirectory )
myFile = io.open( path, "w+b" )
```

In this version of the app, we make a GET request using http.request and provide the sink to handle writing the data to myFile, which is opened in the section above. Finally, the image is displayed from the system.DocumentDirectory where it was saved.

```
local function loadButtonPress (event)
  native.setActivityIndicator( true )

   -- Request remote file and save data to local file
   http.request{
      url =
"http://www.BurtonsMediaGroup.com/MobileAppDevelopmentCover.png",
     sink = ltn12.sink.file(myFile),
   }
   print("should be downloaded now")
    testImage = display.newImage( "MobileAppDevelopmentCover.png",
system.DocumentsDirectory, 10,30);
    testImage.x = display.contentWidth/2
    native.setActivityIndicator( false )

end

-- Load Button
loadButton = ui.newButton{
      default = "buttonBlue.png",
      over = "buttonBlueOver.png",
      onPress = loadButtonPress,
      text = "Load Picture",
      emboss = true
}
loadButton.x = display.contentWidth/2
loadButton.y = display.contentHeight - 50
```

Tracking Network Status

Networking with a mobile device can be a challenge. Users walk in and out of range of wireless connections or switch cell networks on a regular basis; thus it is critical that you build a framework that anticipates these potential networking issues. The following are some tools available to help you manage network interactions:

- network.canDetectNetworkStatusChanges – returns true if the current platform supports status changes to network.
- network.setStatusListener(URL, listener) - monitors a host to see if it can be reached through the network. Monitors the host instead of the hardware. Possible events include:
  - event.address – URL of the event. Useful if you are monitoring multiple hosts.
  - event.isConnectionRequired – returns True Boolean if the connection is up.
  - event.isConnectionOnDemand – returns True Boolean if the connection will come up automatically.
  - event.isInteractionRequired – returns True Boolean if the user needs to enter a password.
  - event.isReachable - returns Boolean if the host can be reached.
  - event.isReachableViaCellular – returns True Boolean if the connection occurs through cellular.
  - event.isReachableViaWiFi – returns True Boolean if the connection occurs through WiFi.

Uploading to a Webserver

There are typically two types of situations that require an upload to a remote server:

1. Uploading form or reponse information to a webserver (which is typically handled by a script file written in a langauge such as PHP).
2. Uploading an image or a file to be stored or shared.

3-Tier Architecture

In both of these cases you will perform a "POST" instead of a "GET". Both of the samples that I am including are from the Ansca Mobile website. I am adding my own explanation on the structure and reasoning for using each of these methods. Both examples take advantage of the standard 3-tier method of network communications, which is what I encourage my students to use for any network-enabled project that they undertake: Client – Server – Database.

The **client** refers to the app that we are creating. The client's only responsibility is to handle the data on the local device, whether that be to display it or capture it for upload to the remote server.

Server is a script on a remote webserver that handles all interaction between the client and the remote database. The server script provides an abstraction layer that improves security (assuming the script is built appropriately), performance, and simplifies data exchange from various possible clients. Usually the server script is written in a scripting language such as PHP.

Database is where we have all of our important data stored on a remote server. While the database can be stored on the same server as the server-script, for greater security or data-intensive apps it is often a good idea to place the database on a dedicated server.

Post Example 1: Uploading Form Data

In the first case the response information can be sent as part of the URL data, making it very easy to handle. First, a network listener function is created to handle the possibility of receiving an error or to handle the response from the POST.

```
local function networkListener( event )
    if ( event.isError ) then
       print( "Network error!")
    else
       print ( "RESPONSE: " .. event.response )
    end
end
```

For the example, postData is set with the message to the php script. It is important that this data contain no spaces. Use the '&' sign between multiple variables to be passed.

```
postData = "color=red&size=small"
local params = {}
```

To complete the POST operation, we set the params.body to the postData and make the network request. In this instance, the call is to the local machine (127.0.0.1 always talks to the local host, in this case the mobile device making the call). Make sure you substitute the appropriate URL and file name to be accessed by the POST operation.

```
params.body = postData
network.request( "http://127.0.0.1/formhandler.php", "POST",
networkListener, params)
```

Post Example 2: Uploading Files or Images

Our second example shows how to upload a JSON (though a JSON file isn't attached). In this example, we begin by setting the header and body information. If you were attaching an image or additional data such as a JSON file, you would set the body equal to the image.png file or the data file. Note that this example also needs the network listerner function just as the first example.

```
headers = {}
headers["Content-Type"] = "application/json"
headers["Accept-Language"] = "en-US"
body = "This is an example request body."
local params = {}
```

```
params.headers = headers
params.body = body
network.request( "https://192.168.0.1/getData.php", "POST",
networkListener,  params)
```

Connecting to Propritary Networks

With the popularity of social websites, it is not surprising that many people wish to create apps and games that connect to such sites. While many such connections are basic http requests (such as twitter), other websites have a few special APIs to make the connection a little simpler.

Facebook

The facebook library (i.e., you have to do a require ("facebook") if it is a library) contains a number of functions to connect to Facebook.com through the Facebook Connection interface. With these functions you will be able to login (or out), post messages and images, as well as retrieve current status. At the time of this writing, the Facebook API requires a device build and is not available through the Corona Simulator.

- facebook.login(appId, listener [, permissions]) – prompts the user to login to facebook. Parameters include:
    - appId – application id supplied by Facebook when you register your application.
    - listener – can be a function or table, but must be able to respond to "fbconnect" events.
    - permissions – an array of strings for Facebook's publishing permissions
    Returns:
    - event.name – the name of the event ("fbconnect")
    - event.type – the type of event ("session")
    - event.phase – the current status: "login", "loginFailed", or "loginCancelled"
- facebook.logout( ) – as you might have guessed, logs the user out of their Facebook account. Returns "logout" in the event.phase
- facebook.request(path [, httpMethod, params]) – Used to GET or POST data to the logged-in Facebook account. Can be used for posting messages and photos as well as getting user data and recent posts.
    - path – the Facebook API graph path: "me", "me/friends", "me/feed", etc
    - httpMethod – "GET" or "POST"
    - params- a table based upon Facebooks API arguments.
    Returns:
    - event.name– name of the event ("fbconnect")
    - event.type – type of event ("request")
    - event.response – theJSON response from Facebook

  - event.isError – True is an error occurred
- facebook.showDialog(params) – display the Facebook dialog for publishing posts to the users status. Simplifies posting updates without having to create a dialog box. Parameters are based upon the Facebook arguments.

Facebook Example

I have included the standard example for a Facebook connection. To begin we will need to load the Facebook library with a require. Next, a listener function is needed to handle the response from the Facebook servers. If the connection is successful, "session" is returned from the server, then we check the current event phase. If "login" is returned, a request of "me/friends" is made. If this request is successful, then a scrolling list of friends' names is created.

```
local facebook = require "facebook"

-- listener for "fbconnect" events
local function listener( event )
   if ( "session" == event.type ) then
       -- upon successful login, request list of friends of
       -- the signed in user
       if ( "login" == event.phase ) then
           facebook.request( "me/friends" )
       end
   elseif ( "request" == event.type ) then
       -- event.response is a JSON object from the FB server
       local response = event.response
        -- if request succeeds, create a scrolling list of friend
names
        if ( not event.isError ) then
            response = json.decode( event.response )
            local data = response.data
            for i=1,#data do
                local name = data[i].name
                print( name )
            end
       end
    elseif ( "dialog" == event.type ) then
        print( "dialog", event.response )
    end
end
```

Now that we have handled the listener, we can make the initial call to Facebook using your app id (provided by Facebook. See http://developers.facebook.com/setup for more information).

```
local appId =  "YOUR FACEBOOK APP ID"
facebook.login( appId, listener, {"publish_stream"} )
```

Papaya and OpenFeint

The gameNetwork library provides access to third party resources for social gaming. This allows you to easily include leaderboards, achievements, challenges, highscore, and many other features. These features are rapidly evolving in what is available on iOS and Android devices, so I recommend that you check the Ansca Mobile website to find the current status on these libraries: http://developer.anscamobile.com/reference/index/game-network. In the mean time, here is a brief introduction into Papaya and OpenFient:

- gameNetwork.init(providerName [, parms ...]) – initializes an app to connect to a game network provider such as OpenFient or Papaya.
- gameNetwork.request(command [, parms ...]) – send or request information from the game network provider.
- gameNetwork.show(name [, data]) – Displays information from game network on the screen.

Papaya Example

Papaya provides social networking resources for your mobile apps. You will need to get a key for your app from http://www.papayamobile.com, which will allow interaction across Papaya's broad array of social network games. You can use Papaya for leaderboards, achievements, score boards, etc. At the time of this writing, Papaya is for the Android side of social networking.

```
local gameNetwork = require "gameNetwork"

gameNetwork.init( "papaya", "papayaSocialKey" )
gameNetwork.request( "setHighScore",{ leaderboardID = "Level1", score
= 321 })
gameNetwork.request( "unlockAchievement", " 184 " )
```

OpenFeint Example

OpenFeint provides the iOS side of social networking for your app. You can use it for leaderboards, achievements, scoreboards, and other standard social networking features. Go to http://www.openfeint.com/developers to learn more about the many OpenFeint features. At the time of this writing, OpenFeint is only available for iOS.
OpenFeint does provide an interesting save feature that allows you to save game data to the cloud as a blob of data.

```
local gameNetwork = require "gameNetwork"

gameNetwork.init( "openfeint", "product-key", "secret", "display
name", "appId" )
gameNetwork.request( "setHighScore", { leaderboardID="abc123",
score=99, displayText="99 sec" } )
gameNetwork.request( "unlockAchievement", "achievementId" )
```

```
gameNetwork.request( "uploadBlob", key, data )

-- listener for "completion" event with "blob" key set.
gameNetwork.request( "downloadBlob", key, listener )

gameNetwork.show( "highscore", "abc123" )
```

inMobi

At the time of this writing, Corona supports one ad vendor; inMobi. To include ads from inMobi, you must have an application key (provided by inMobi when you sign-up for an account). You can register with inMobi at http://www.inmobi.com.

To include ads in your app, you will need require ads:
local ads = require "ads"

API calls for ads include:

- ads.init(providerName, appId) – Initializes the Ads library.
- ads.hide() – stop showing ads.
- ads.show(adUnitType [, {x=0, y=0, interval = 5, testmode=false}]– Display ads at given screen location with a specified refresh time.
  adUnitType:
  - "banner320x48"
  - "banner300x250"
  - "banner728x90" (iPad only)
  - "banner468x60" (iPad only)
  - "banner120x600" (iPad only)

  Parameters:
  - x, y – Left, Top corner of banner position. Defaults to 0.
  - interval – ad refresh time in seconds. Defaults to 10.
  - testmode – For testing. Default is false.

inMobi Example

```
local ads = require "ads"
ads.init( "inmobi", "YourAppID" )

-- for  iPhone, iPod Touch, iPad
ads.show( "banner320x48", { x=0, y=100, interval=5, testMode=false } )
```

Virtual Currency Credits

While I personally do not like these types of services, I realize that many people do, or there wouldn't be such a demand. The credits library allows your app to provide offer-based virtual currency, enabling the users of your apps to participate in advertisement offers in exchange for virtual currency. Credits uses the Super Rewards service to offer users of your credit to purchase in-app items, levels, etc in exchange for participating in these offers.

The credits API includes:

- credits.init(appId, [, uid], listener) – Starts Super Rewards API in the app.
- credits.requestUpdate() – http request to return number of new and total credits. Sends the results to the listener specified in credit.init.
- credits.showOffers()- Displays a scrolling offers list that users can select from or dismiss.

```
local credits = require "credits"

local creditsListener = function( event )
   print( event.name ) -- outputs "creditsRequest"
   if ( event.isError ) then
        print( "An error occurred. Request failed" )
   else
        print( event.newCredits ) --number of new credits
        print( event.totalCredits ) --total available credits    end
end

credits.init( "yourAppIdHere", creditsListener )
credits.showOffers()
credits.requestUpdate()
```

To use Credits, you wil need to activate your account with SuperRewards by going to http://developer.anscamobile.com/credits.

Pubnub

Pubnub is a nice little API that allows client-to-server or client-to-client communications. It makes creating a multi-user app very easy. It is free to use for development (by using the key "demo") and in use allows up to 5000 messages per day (each send or receive counts as a message). After the 5000 usage, the cost is $0.0001 per message, volume discounts are available.

Project 14.1 Multi-User App

To demonstrate pubnub, we are going to create a simple demo app that when the box on the screen is moved, it will automatically move on any other device currently running the program. To get started, our build.settings and config.lua files are pretty standard:

build.settings

```
settings =
{
	androidPermissions =
	{
		"android.permission.INTERNET"

	},

	orientation =
	{
		default ="portrait",
		supported =
		{
	"landscapeLeft","landscapeRight","portrait","portraitUpsideDown"
		},
	},

}
```

config.lua

```
application =
{
	content =
	{
		width = 320,
		height = 480,
		scale = "letterbox",
		fps = 30,
		antialias = false,
		xalign = "center",
		yalign = "center"
	}
}
```

main.lua

The main.lua file will begin by loading physics and pubnub, start the physics engine and set the gravity to zero. I am using the same drag routine that we previously used in chapter 10 to move the starship.

Then we will create a box, place it in the center of the screen, and add the box as a physics body.

```
local physics = require "physics"
require "pubnub"
physics.start()
physics.setGravity(0,0)

-- create box to move on screen
local box = display.newImage("button1.png")
box.x = display.contentWidth/2
box.y = display.contentHeight/2
physics.addBody(box, {friction = 0, bounce = 0, density = 0})
```

Now we will initialize the pubnub network. As this is a development project, I am using the publish key and subscribe key of demo. When it is time to create your own project for publication, you will need to secure keys from pubnub for your app.

```
-- intialize pubnub networking
multiplayer = pubnub.new({
    publish_key    = "demo",
    subscribe_key = "demo",
    secret_key     = nil,
     ssl              = nil,
     origin          = "pubsub.pubnub.com"
})
```

Now let’s setup everything to receive push messages. The channel should be a unique name that is only used by the instance of the game or app. The callback function will handle any push notifications: in this case it receives the message which contains the new x and y position for the box, which we use with a transition.to command.

```
multiplayer:subscribe({
    channel = "Ch14-MultiUserDemo",
    callback = function(message)
    print("Received: "..message.msgtexta..", "..message.msgtextb)
    transition.to(box, {x=message.msgtexta, y=message.msgtextb, timer
= 500})
    end,
    errorback = function()
        print("Oh no!!! Dropped 3G Conection!")
    end
})
```

The send a message function receives the boxes x and y location and publishes it to the same channel as above. Within the message array you can send any data necessary for app or game play.

```
function send_a_message(imageX, imageY)
    multiplayer:publish({
```

```
        channel = "Ch14-MultiUserDemo",
        message = { msgtexta = imageX, msgtextb=imageY }
    })
end
```

The update coordinate function is called every second and sends the current x and y coordinates of the box.

```
function update_coord()
    send_a_message(box.x, box.y)
end
```

Now the drag function, which is the same function we used in the star explorer example in chapter 10. Note that there are other methods that can be used for the drag function that allow for rotation.

```

local function startDrag( event )
   local t = event.target

   local phase = event.phase
   if "began" == phase then
     display.getCurrentStage():setFocus(t)
     t.isFocus = true

     --Store inital position
     t.x0 = event.x - t.x
     t.y0 = event.y - t.y

-- make the body type 'kinematic' to avoid gravity problems
     event.target.bodyType = "kinematic"

     -- stop current motion
     event.target:setLinearVelocity( 0,0)
     event.target.angularVelocity = 0

   elseif t.isFocus then
     if "moved" == phase then
          t.x = event.x - t.x0
          t.y = event.y - t.y0
      elseif "ended" == phase or "cancelled" == phase then
          display.getCurrentStage():setFocus(nil)
          t.isFocus = false

          -- switch body type back to "dynamic"
          if (not event.target.isPlatform) then
               event.target.bodyType = "dynamic"
          end
     end
   end
     return true
```

```
end
```

Finally, we will use a timer that will make a call to update_coord once per second for 100 seconds. We also add an event listener to handle the drag event.

```
timer.performWithDelay( 1000, update_coord, 100 )

box:addEventListener("touch", startDrag)
```

You probably noticed that the send and receive channel are the same for this project. This does have the effect that the app will receive its own data. If you are making a head to head game, it would be better if you receive on one channel and send on another, so that player 1 sends on the same channel that player 2 receives and player 2 sends on the channel that player 1 receives. This will reduce the number of send and receive messages generated by your app.

If you are not concerned about the number of messages you are generating, then you could also use a player id system so that any message that is sent by a player will be disregarded if received by the same player.

Summary

In this chapter, we have examined how to setup basic communications with a webserver, proprietary services, or with another app. Hopefully you will find these resources a great starting point as you create your network-enabled app.

Assignments

1) Modify Project 14.1 so that a send only occurs when the box has been moved.

2) Create a simple multiplayer table tennis app that is based upon Project 14.1.

3) Create a messaging app using pubnub to send messages to your friends.

4) Challenge: Create a secret code to encode the message in assignment 3 so that the reader must enter the right decode password to read the message.

5) Modify 14.1 so that apps only receive messages that they did not send.

Chapter 15:
Working with Widgets & Popups

Widgets are one of the most recent additions to the Corona API. While still in beta at the time of this writing, they are so important to app development that I felt it would be remiss to not include them. Specifically in this chapter we will:

- Introduce the use of widgets in app development
- How to use widget themes
- Review the use of the different types of widgets
- How to properly remove a widget
- Introduce the use of Web Popup

Widgets

The version of widget that I will be discussing in this chapter is 0.2, which is still in beta. By the time you read this there are likely to be updates and additional features or bug fixes available.

Widgets provide user-interface tools that are standard features when programming in the native development environment for iOS and Android. With widgets, you can create apps that include native features such as a picker wheel or slider, but take a fraction of the time to develop.

You should not consider a widget a typical display object. While they can be included in groups, they must be inserted by their view property:

```
myGroup:insert(myWidget.view)
```

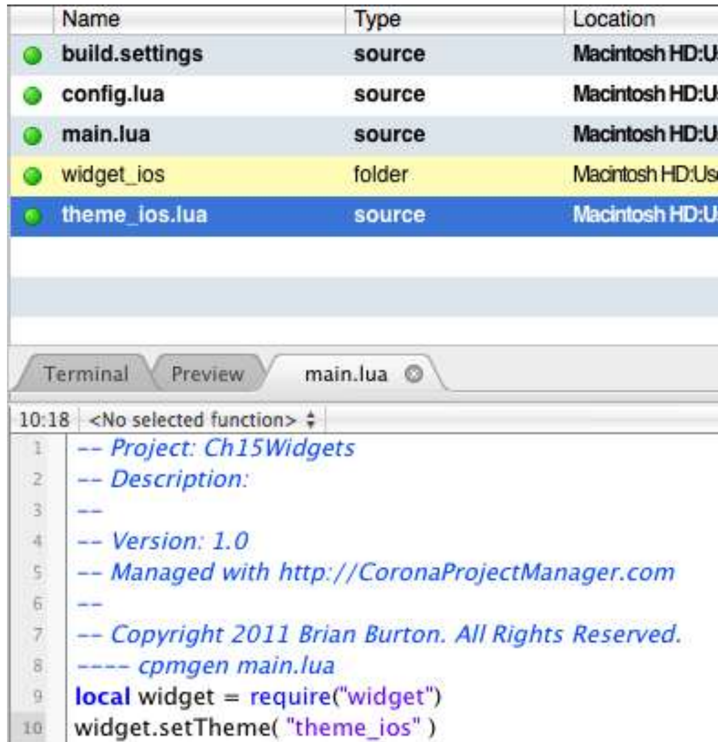
Widget Themes

The widget.setTheme allows you to give the widgets a specific look and feel. A list of current themes can be found at:
http://developer.anscamobile.com/content/widget#Using_Widget_Themes.
Themes currently simulate the iOS or Android basic appearance. The theme should be set immediately after the require widget command. Themes are Lua files with theme tables that correspond to each widget. The file widget_ios or widget_android (not available at the

time of this writing) as well as the various theme folders must be in your project folder. Both the slider and picker wheel widgets are best used with themes.

widget.setTheme Example

```
    local widget = require "widget"
    widget.setTheme( "theme_ios" )
```



File/folder list and first few lines of code as seen from Corona Project Manager

widget.newButton

The first widget will be easy to get use to. You are already use to using something similar to it through the UI library.
The widget.newButton provides a button that supports onPress, onRelease, and onDrag events.

Parameters:

- id – an optional string that can be used to identify the button (default is nil).
- left, top – intial coordinates of buttons left, top corner.
- width, height – allows adjusting the buttons width and height (default is width = 124, height = 42).
- label – text that will appear on the button
- labelColor – RGBA table showing default and over color states of the label text
- offset – adjust the y axes of the label text
- font – allows changing the button label font (default is native.systemFont)

- fontSize – the label font size in pixels (default is 14).
- emboss - Boolean that will allow the text to appear embossed.
- default, over – image files to represent different states of the button. If no image is specified and there is no theme, the button will default to a rounded rectangle.
- baseDir – base directory for custom images (default = system.ResourceDirectory – the project folder).
- defaultColor, overColor – tables to hold the RGBA of the default and over states if there is no custom image.
- strokeColor – RGBA table for the button's border if there is no custom image in use.
- strokeWidth – width fo the border of the button, if there is no custom image in use.
- cornerRadius – controls the curve of the default rounded rectangle button (default is 8 pixels).
- onPress – callback function for when the button is tapped.
- onRelease – optional callback function that is called when the user ends the tap/press of the button.
- onDrag – optional callback function for a user drag event on the button.
- onEvent – optional function and should only be used if none of the other above events are used.

Properties:

- view – allows the widget to be added to a group display.
- x, y – moves the object to a new location on the display.

Methods

- setLabel(*string*) – Changes the button's label text.
-

widget.newButton Example
Adding a button should look very familiar from your use of the UI.lua library. Do note that you can use the setTheme to make your buttons look more typical for your target operating system.

```
local widget = require("widget")
widget.setTheme("widget_ios")
    local myButton = widget.newButton{
        id = "button1",
        left = 100,
        top = 200,
        label = "myWidget Button",
        width = 150, height = 28,
        cornerRadius = 8,
        onPress = myButtonPress
    }
```

widget.newTabBar

The widget.newTabBar allows you to create a customizable tab bar. Tabs are auto-positioned based upon the number of buttons. While there is no limit to the number of buttons that can be added, do remember that the button does need to be large enough to tap. The widget itself only has a few parameters:

Parameters:

- width, height – allows custom width and height of tab bar (default width is display.contentWidth, height is 50)
- left, top – allows custom location of tab bar (default is bottom of the screen)
- background – set a static image for background of tab bar.
- topGradient – allows custom gradient using graphics.newGradient(). If no background is set, default will be a gradient for top-half of tab bar and solid bottom.
- bottomFill – table of RGBA for bottom of tab bar if background is not set.
- buttons – table holding the parameters and options for each tab button (see Buttons Table).

Buttons Table

- id – string to identify button (default is button's index).
- up – icon filename for 'up' state of button.
- upWidth, upHeight – width and height of up icon. Recommend size is 32 x 32.
- down - icon filename for 'down' state of button.
- downWidth, downHeight – width and height of down icon. Recommended size of 32x32.
- label – text that will appear on the button below the icon.
- labelColor – RGBA table showing default and over color states of the label text
- font – allows changing the button label font (default is native.systemFontBold)
- size – the label font size in pixels (default is 10).

- baseDir – base directory for custom images (default = system.ResourceDirectory – the project folder).
- cornerRadius – controls the curve of the default rounded rectangle button (default is 8 pixels).
- onPress – callback function for when the button is tapped.
- selected – Boolean to track if the button is selected (down). Only one button may be down at a time.

  Properties:
- view – allows the widget to be added to a group display.
- x, y – moves the object to a new location on the display.

The onPress event listener can receive the Event Table that is passed by the event. The table includes:

- event.name – will always be "tabButtonPress"
- event.target – reference to tab button that triggered event
- event.targetParent – reference to tab bar widget that triggered the event

widget.newTabBar Example

This should look fairly familiar from your work with buttons in the UI.lua library. Each of the tab bar buttons acts as a button.

```
local widget = require "widget"

local function onBtnPress( event )
print( "You pressed tab button: " .. event.target.id )
end

local tabButtons = {
        {
            label="First Tab",
            up="firstIcon.png",
```

```
            down="firstIcon-down.png",
            width=32, height=32,
            onPress=onBtnPress,
            selected=true
        },
        {
            label="Second Tab",
            up="secondIcon.png",
            down="secondIcon-down.png",
            width=32, height=32,
            onPress=onBtnPress
        },
    }

local tabs = widget.newTabBar{
        top=430,
        buttons=tabButtons
    }
```

widget.newSlider

The widget.newSlider allows you to create a slider object that can be adjusted in width. This widget is very similar to Apple's iOS slider. It is also very flexible and includes quite a few parameters:

Parameters:

- id – string to identify button (default is button's index).
- x – starting x value (default is 0).
- y – starting y value (default is 0).
- width – horizontal length of the widget (default is 220).
- value – sets or returns value of the slider between 0 and 100 (default is 50).
- callback – listener function called every time the slider is touched or moved.
- leftImage ({*imageFile, imageWidth, imageHeight, baseDir}*) – table object that represents the left-edge of the slider.
- rightImage({*imageFile, imageWidth, imageHeight, baseDir}*) – table object that represents the right-edge of the slider.
- maskImage({*imageFile, imageWidth, imageHeight, baseDir}*) – table object that represents the bitmap mask of the slider.
- fillImage({*imageFile, imageWidth, imageHeight, baseDir}*) – table object that represents the fill of the slider.
- handleImage({*imageFile, imageWidth, imageHeight, baseDir}*) – table object that representsthe handle of the slider.

Properties:

- view – allows the widget to be added to a group display.
- x, y – moves the object to a new location on the display.

widget.newSlider Example

To implement the slider widget, you will first need to load the widget library with the require command. Unless you want to go through and set all of the various images (which you are welcome to do), I recommend using the pre-defined themes. Make sure that the appropriate widget theme folders are available for your app and that widget_ios.lua is copied into your folder.

Be sure to set your listener function for your slide event. You can also check for phases "moved" and "released". Moved is a response to the current movement of the slider. Released is the response to when the slider event is completed. In the example below, the listener prints any movement of the slider or either phase.

```
local widget = require("widget")
widget.setTheme( "theme_ios" )

-- Callback listener for slider widget:
local sliderListener = function( event )
    local sliderObj = event.target
    print( "New value is: " .. event.target.value )
end

-- Create the slider widget
local mySlider = widget.newSlider{
    width=160,
```

```
    callback=sliderListener
    }

-- Center the slider widget on the screen:
mySlider.x = display.contentWidth * 0.5
mySlider.y = display.contentHeight * 0.5

-- adjust the slider width:
mySlider.width = 240

-- set the value manually:
mySlider.value = 75

-- insert the slider widget into a group:
someGroup:insert( mySlider.view )
```

widget.newTableView

The tableView widget allows you to create scrolling lists. With this widget you can control the rendering of the individual rows. The table view offers another option of how to display tables beyond what was discussed in chapter 13. The tableview widget is very powerful and flexible; with that flexibility comes a few more properties and methods:

Parameters:

- bgColor – RGBA table to set the color of the rectangle that is behind the tableView (default is white {255, 255, 255, 255}).
- width, height – allows custom width and height of table view (default width is full width and height of the screen).
- left, top – allows custom location of table view (default is 0)
- topPadding, bottomPadding – number of pixels from the top and bottom in a tableView where rows will stop when you reach the top or bottom of the list (default is 0).
- friction – determines how fast the rows travel when flicked up or down (default is 0.935).
- maskFile – allows a custom height.
- renderThresh – pixel amount to the top and bottom of the table view in which rows are rendered and de-rendered (default is 150)

Properties:

- view – allows the widget to be added to a group display.
- view.content – display group that represents the objects of each row. The row data table can be accessed from view.content.rows.
- isLocked – Boolean that will prevent scrolling if true.
- tableView:getScrollPosition() – returns the current y position of the table view content. Used to mark current location.
- tableView:scrollToY(y postion, time) –scroll to specified y position. Time is in milliseconds for how long it takes to scroll to location (default time is 1500).

- tableView:scrollToIndex(index, time) – scrolls table to specific row. Time is in milliseconds for how long it takes to scroll to the location (default time is 1500).
- tableView:insertRow ({params}) – used to insert rows into the table view. Accepts as parameters:

  insertRow Paramerters:
  - width, height – allows adjustment of individual rows
  - rowColor – RGBA table to set row color.
  - lineColor – RGBA table to set the separator line color.
  - isCategory – Boolean specifiying the current row as a category.
  - onEvent – function callback for events.
  - onRender – function callback for creating the row's visual elements.
- tableView:deleteRow (row or index) – deletes a specific row.

  Row Events – when insertRow method is called the following keys are passed as part of the event table:
- event.name – either tableView_onRender (for onRender listeners) or tableView_onEvent (for onEvent listeners).
- event.tableView – reference to the calling tableView object.
- event.target – reference to the calling row that triggered the event.
- event.view – reference to the display group. If you create a display object for a specific row in your onRender listener function, you MUST insert those objects in to the event.view group or they will not render properly and may cause memory leaks.
- event.phase – will either be "press" or "release". You should always test for the phase for onEvent listener functions, and return true on success.
- event.index – number that represents the row's position in the table view.

widget.newTableView Example

After the required widget, we begin this example with specifying the tableview options. I have included the mask-410.png file in download files. By setting the height to 410 pixels and including the mask we are able to leave the bottom portion of the screen open. The top of the table is set to begin at the bottom of the status bar.

```
local widget = require "widget"

local tableOptions = {
        top = display.statusBarHeight,
        height = 410,
        maskFile = "mask-410.png"
        }

local list = widget.newTableView( tableOptions )
```

The event listener captures the row specified by the event, changes the row's alpha to .5 while being pressed and prints to the terminal window the row tapped upon release.

```
-- onEvent listener for the tableView
local function onRowTouch( event )
    local row = event.target
    local rowGroup = event.view

    if event.phase == "press" then
       if not row.isCategory then rowGroup.alpha = 0.5
       end
    elseif event.phase == "release" then
       if not row.isCategory then
       -- refresh if still onScreen when content moves
             row.reRender = true
             print( "You touched row #" .. event.index )
       end
    end

    return true
end
```

The onRowRender function handles creating the row for the display as the user scrolls through the list.

```
-- onRender listener for the tableView
local function onRowRender( event )
    local row = event.target
    local rowGroup = event.view

    local text = display.newRetinaText( "Row #" .. event.index, 12, 0,
"Helvetica-Bold", 18 )
    text:setReferencePoint( display.CenterLeftReferencePoint)
    text.y = row.height * 0.5
    if not row.isCategory then
```

```
        text.x = 15
        text:setTextColor( 0 )
    end

    -- must insert everything into event.view:
    rowGroup:insert( text )
end
```

Now we will create the row information. In the 25th and 45th rows, categories are created. The final portion of the code handles inserting the row and setting the events and render handlers.

```
-- Create 100 rows, and two categories to the tableView:
for i=1,100 do
   local rowHeight, rowColor, lineColor, isCategory

   -- make the 25th item a category
   if i == 25 then
      isCategory = true; rowHeight = 24; rowColor={ 70, 70, 130, 255
}; lineColor={0,0,0,255}
   end

   -- make the 45th item a category as well
   if i == 45 then
       isCategory = true; rowHeight = 24; rowColor={ 70, 70, 130, 255
}; lineColor={0,0,0,255}
   end

   -- function below is responsible for creating the row
   list:insertRow{
          onEvent=onRowTouch,
          onRender=onRowRender,
          height=rowHeight,
          isCategory=isCategory,
          rowColor=rowColor,
          lineColor=lineColor
        }
end

-- delete the tenth row in the tableView
list:deleteRow( 10 )
```

If you want to go to a detail view or a secondary table, just call the detail/second table function from within the onRowTouch function, just like we did in chapter 13.

widget.newScrollView

The scroll view widget allows you to create scrolling content areas. It should be noted that if you want a scrollview that does not extend the full height of the screen, you will need to

create a bitmap mask that is the width and height of the scrollview that you desire for your app. The scrollview is a fairly straight forward widget with only a few parameters:

Parameters

- left, top – allows custom location of scrollview (default is 0 for both values)
- width, height – allows custom width and height of scroll view (default is full width and height of the screen).
- topPadding, bottomPadding – number of pixels from the top and bottom in a tableView where rows will stop when you reach the top or bottom of the list (default is 0).
- friction – determines how fast the rows travel when flicked up or down (default is 0.935).
- maskFile – allows a custom height.

Properties:

- view – allows the widget to be added to a group display.
- view.content – display group that represents the objects of each row. The row data table can be accessed from view.content.rows.
- scrollView:getScrollPosition() – returns the current y position of the scrollView content. Used to mark current location.
- scrollView:scrollToY(y postion, time) –scroll to specified y position. Time is in milliseconds for how long it takes to scroll to location (default time is 1500).
- insert() – add items to scrollView.

widget.newScrollView Example

```
local widget = require "widget"

-- Create a new ScrollView widget:
local scrollView = widget.newScrollView{ height=320, maskFile="mask-
320x320.png" }

-- Create an object and place it inside of ScrollView:
local myObject = display.newImage( "myobj.png" )
scrollView:insert( myObject )

-- Place the ScrollView into a group:
local someGroup = display.newGroup()
someGroup:insert( scrollView.view )

-- Remove the ScrollView:
display.remove( scrollView )
scrollView = nil
```

widget.newPickerWheel

The picker wheel widget allows the user to rotate a dial to select their response for the application. This widget has a great deal of flexibility, and with that comes a great number of parameters:

Parameters:

- id – string to identify the picker wheel.
- width, height – sets the corresponding attribute of the picker wheel (default: width = 296, height = 222).
- left, top – top, left corner of the widget.
- totalWidth – width of the background (default is display.contentWidth).
- selectionHeight – height of your selection area graphic (default is 46).
- font – font used when rendering column rows (default is native.systemFontBold).
- fontSize – size in pixels of the font used for rendering the text (default is 22).
- fotnColor – RGBA table for text color of each column (default is black).
- columnColor – RGBA table for column background color (default is white).
- background – sets a background image for the picker wheel.
- backgroundWidth, backgroundHeight – if you specifiy a background image, you must set the width and height of the background image.
- glassFile – filename for the picker wheel's selection area graphic.
- glassWidth, glassHeight – If you have a glassFile, you should set the images width and height.
- separator – a graphic that is used to separate the columns.
- separatorWidth, separatorHeight – separator image width and height.
- maskFile – used to crop columns.
- baseDir – base directory for graphics (default is system.ResourceDirectory).
- columns – table array that will have sub-tables arrays representing the individual columns of your picker wheel.

Column Properties:

- width – sets the column to a custom width (default is all columns are equal width).
- startIndex – sets the column at a specific row.
- alignment – sets text to left, right, or center alignment (default is left).

Properties:

- view – allows the widget to be added to a group display.
- x, y – moves the object to a new location on the display.
- picker:getValues() – returns a table holding the value/index of the rows that are currently selected.

widget.newPickerWheel Example

In this example, a 3-column time picker wheel is created. We will be using an array within an array. First create the ColumnData array, which will have 3 columns. Next create a 12-row array for the first column that represents the hour.

```
local widget = require "widget"
widget.setTheme( "theme_ios" )
-- create table to hold all column data
local columnData = {}

-- create first column
columnData[1] = { "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
"11", "12" }
columnData[1].alignment = "right"
columnData[1].width = 120
columnData[1].startIndex = 7
```

Now we will populate the second column, which represents minutes, using a for loop. The for loop allows us to create the 0 to 59 elements of the array efficiently.

```
-- second column (Populated for minutes)
columnData[2] = {}
columnData[2].alignment = "center"
for i=0,59 do
   columnData[2][i] = i
end
columnData[2].startIndex = 30
```

The third column will represent night or day, so will only have two elements.

```
-- third column (Select AM or PM)
columnData[3] = { "AM", "PM" }
columnData[3].startIndex = 2
```

The last step in creating a picker wheel is the actual picker wheel call.

```
-- create the actual picker widget with column data
local picker = widget.newPickerWheel{
        id="myPicker",
        font="Helvetica-Bold",
        top=258,
        columns=columnData
}
```

Removing Widgets

Since widgets are not typical display objects, you must remove widgets manually. You can only use the display.remove() or removeSelf() methods to delete a widget from view. To avoid memory leaks in your program, you must first manually remove any widgets before removing any group that they might be associated.

```
display.remove(myWidget)
myWidget = nil

display.remove (someGroup)
someGroup = nil
```

This will ensure that memory is conserved as well as preventing your app from crashing.

Project 15: Longitude and Latitude

This sample project is designed to allow the user to select a country, state (if appropriate) and city via a picker wheel. Once these pieces of data are entered, the app will return the longitude and latitude of the location. I do wish to make it known that I am aware that countries beyond the United States and Canada have states and providences; however, this data was not available at the time of building this app. So please understand that no slight to other nations is intended.

This is an early version of the app. A full version that includes maps, GPS, advertisements and the ability to store locations is included in my next book “More Mobile App Development with Corona.”

Before you begin this project, make sure you have the theme_ios.lua and widget_ios folder in your project folder.

config.lua

```
application =
{
	content =
	{
		width = 320,
		height = 480,
		scale = "letterbox",
		fps = 30,
		antialias = false,
		xalign = "center",
		yalign = "center",

		imageSuffix =
		{
			["@2"] = 2
		}

	}
}
```

build.settings

```
settings =
{
	android =
	{
		versionCode = "1.0"
	},
	orientation =
	{
		default ="portrait",
		content = "portrait",
		supported =
		{
			"portrait"
		},
	},
}
```

main.lua

The database that was used for this app is a modified db supplied for free by MaxMind: however, MaxMind, Inc. requires the following statement to accompany the data:

```
-- locations database is derived from MaxMind World Cities database
and is Copyright 2008 by MaxMind Inc.
--[[ Redistribution and use with or with out modifications, are
premitted provided that the following conditions
are met:
1. Redistributions must retain the above copyright notice, this list
of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.
2. all advertising materials and documentation mentioning features or
use of this database must display the following acknowledgement:
 "This Product includes data createdby MaxMind, available from
http://www.maxmind.com/"
3. "MaxMind" may not be used to endorse or promote products derived
from this database without
specific prior written permission.]]
.
```

To get things started, we will need widget and sqlite for this project. I am using the iOS theme provided by Ansca Mobile. After initializing the needed variables, we'll copy the database over to the documents directory as we have done previously.

```
local widget = require "widget"
local sqlite = require "sqlite3"

widget.setTheme( "theme_ios" )

local columnData = {}
```

```
local picker, selectButton, selectedCountry, selectedState,
selectedCity, LongCN, LongState

-- Does the database exist in the documents directory (allows updating
and persistance)
local path = system.pathForFile("locations.sqlite",
system.DocumentsDirectory )
file = io.open( path, "r" )
   if( file == nil )then
      -- Doesn't Already Exist, So Copy it In From Resource Directory
      pathSource     = system.pathForFile( "locations.sqlite",
system.ResourceDirectory )
      fileSource = io.open( pathSource, "r" )
      contentsSource = fileSource:read( "*a" )
           --Write Destination File in Documents Directory
           pathDest = system.pathForFile( "locations.sqlite",
system.DocumentsDirectory )
           fileDest = io.open( pathDest, "w" )
           fileDest:write( contentsSource )
            -- Done
           io.close( fileSource )
           io.close( fileDest )
   end
-- One Way or Another The Database File Exists Now -- So Open Database
Connection
local db = sqlite.open( path )
print ("Locations opened")

-- handle the applicationExit event to close the db
local function onSystemEvent( event )
      if( event.type == "applicationExit") then
           db:close()
      end
end
```

Now that the database is open we can load our first set of data into an array. After loading the data, we will configure the column properties. Initially only one column will be showing, so we can use the full width of the picker wheel. I have arbitrarily set the startIndex location to center on the United States (only because that's where I am located, feel free to change it!).

```
-- load first column data from SQLite database
  columnData[1]={}
  countryData={}
  local count =0
  local sql = "SELECT DISTINCT country, cn FROM Country ORDER BY
country ASC"
  for row in db:nrows(sql) do
       count = count +1
       columnData[1][count]= row.country
       countryData[count]=row.cn
       -- print(columnData[1][count])
  end
  columnData[1].alignment = "left"
  columnData[1].width = 296
   columnData[1].startIndex = 180
   columnData[1].fontSize = 12
```

The myCityButtonPress function is the last function called in the app, but, as you have seen in the past, dependent functions must be placed above the function that calls them. So we will look at the app in reverse order.

This function starts by storing which row was highlighted by the picker. Calling it the selectedColumn might be a little confusing since it is actually selecting a row of data rather than a column. The picker:getValues returns the current value or index from each column that is highlighted.

```
local myCityButtonPress = function()
    local selectedColumn = picker:getValues()
    local count = 0
```

```
    local sql
```

As I mentioned previously, I do not have state/province data for all of the countries in the database. So I have to compensate in the program by only adding the state/province data if it is available. This creates an interesting challenge in the sql call statement that is easily taken care of with an if then.

```
    if selectedState ~= nil then
        print(selectedColumn[3].index)
        selectedCity = selectedColumn[3].value
        print(selectedCity)
        sql = "SELECT * FROM lonlat WHERE country = 
'"..selectedCountry.."' and state = '"..selectedState.."' and city = 
'"..selectedCity.."' ORDER BY city ASC"
    else
        print(selectedColumn[2].index)
        selectedCity = selectedColumn[2].value
        print(selectedCity)
        sql = "SELECT * FROM lonlat WHERE country = 
'"..selectedCountry.."' and city = '"..selectedCity.."' ORDER BY city 
ASC"
    end
```

After taking care of the state problem, we can pull in the longitude and latitude data. In the database it was necessary to split this data into 3 fields each, which is inconvenient, but easily rectified with a little concatenation.

Finally ,we will display the results to the screen after removing the picker and button from view.

```
    for row in db:nrows(sql) do
```

```
            count = count +1
            lon = row.lon .. "."..row.lond..row.londir
            lat = row.lat.."."..row.latd..row.latdir
            print("Longitude: "..lon..", Latitude: "..lat)
     end
    display.remove(picker)
    picker=nil
    display.remove(selectButton)
    selectButton = nil
    local cn = display.newText("Country: "..LongCN, 20,20)
    cn:setTextColor(255,255,255)
    if selectedState ~= nil then
      local st = display.newText("State: "..LongState,20, 50)
      st:setTextColor(255,255,255)
    end
    local city = display.newText("City: "..selectedCity,20, 80)
    city:setTextColor(255,255,255)
    local long = display.newText("Longitude: "..lon,20,110)
    long:setTextColor(255,255,255)
    local lati=display.newText("Latitude: "..lat,20,140)
    lati:setTextColor(255,255,255)
end
```

The loadCityData function does just that, loads the city data based upon the country and state (if available). The first line of the function resets the button widget onPress event so that when the city is selected, the appropriate function is called. After the data is loaded, the picker wheel is reconfigured for the new data by removing it from the display then adding it back with the new configuration.

```
local loadCityData = function()
     selectButton.onPress = myCityButtonPress
     local count = 0
     -- handle countries with states else
     if selectedState ~= nil then
          columnData[3]={}
          local sql = "SELECT city FROM lonlat WHERE country =
'"..selectedCountry.."' and state = '"..selectedState.."' ORDER BY
city ASC"
         for row in db:nrows(sql) do
               count = count +1
               columnData[3][count]= row.city
              -- print(columnData[3][count])
         end
         display.remove(picker)
        picker = nil
        columnData[1].width= 50
        columnData[2].width = 50
        columnData[2].alignment = "left"
        columnData[3].width= 196
        columnData[2].fontSize = 12
        picker = widget.newPickerWheel{
           top=208,
```

```
            font="Helvetica-Bold",
            fontSize=16,
            columns = columnData  }
      else
            columnData[2]={}
            local sql = "SELECT city FROM lonlat WHERE country =
'"..selectedCountry.."' ORDER BY city ASC"
         for row in db:nrows(sql) do
              count = count +1
              columnData[2][count]= row.city
              print(columnData[2][count])
         end
         display.remove(picker)
         picker = nil
         columnData[1].width = 60
         columnData[2].alignment = "left"
         columnData[2].width= 236
         picker = widget.newPickerWheel{
            top=208,
            font="Helvetica-Bold",
            fontSize=16,
            columns = columnData  }
      end
end
```

You probably noticed in this last section of code that it was necessary to specify the picker wheel in two different configurations depending on whether state data was available or not.

The myStateButtonPress and loadStateData functions are only called if state/providence data had to be loaded. They handle the specifics for which state or providence was selected.

```
local myStateButtonPress = function()
         local selectedColumn = picker:getValues()
         print(selectedColumn[2].index) -- to get the value of the
selected column use .value
          LongState = selectedColumn[2].value
         selectedState = stateData[selectedColumn[2].index]
         columnData[2] = stateData
         columnData[2].startIndex=selectedColumn[2].index
         print(selectedState)
         loadCityData()
end

local loadStateData = function()
         selectButton.onPress = myStateButtonPress
         columnData[2]={}
         stateData = {}
         print(selectedCountry.." in load State Data")
         local count =0
```

```
       local sql = "SELECT state, st FROM State WHERE country =
'"..selectedCountry.."' ORDER BY state ASC"
       for row in db:nrows(sql) do
            count = count +1
            columnData[2][count]= row.state
            stateData[count]=row.st
           --print(columnData[2][count])
       end
       display.remove(picker)
       picker = nil
       columnData[1].width = 60
       columnData[2].alignment = "left"
       columnData[2].width= 236
       picker = widget.newPickerWheel{
           top=208,
           font="Helvetica-Bold",
           fontSize=16,
           columns = columnData  }
end
```

The myCountryButtonPress is the first function called. It is called when the user has selected which country they would like information on.

```
local myCountryButtonPress = function()
        local selectedColumn = picker:getValues()
        print(selectedColumn[1].index) -- to get the value of the
selected column use .value
        LongCN = selectedColumn[1].value
        selectedCountry = countryData[selectedColumn[1].index]
        columnData[1] = countryData
```

```
        columnData[1].startIndex=selectedColumn[1].index
        --print(selectedCountry)
        if (selectedCountry == "US" or selectedCountry == "CA" ) then
           loadStateData()
        else
           loadCityData()
        end
end
```

And finally we make our initial calls to the picker wheel and button widgets and setup the system listener to close the database when the app is closed.

```
picker = widget.newPickerWheel{
        id="myPicker",
        font="Helvetica-Bold",
        fontSize=16,
        top=208,
        columns=columnData
}

selectButton = widget.newButton{
        id = "button1",
        left = 90,
        top = 100,
        label = "Select",
        width = 150, height = 28,
        cornerRadius = 8,
        onPress = myCountryButtonPress
    }

-- system listener for applicationExit
Runtime:addEventListener ("system", onSystemEvent)
```

Web Popups

Web Popups allow you to load a webpage (whether local or remote) from within your app. When called, the web popup will be loaded on top of your current application, filling the entire screen. By default ,the URL is assumed to be the URL of a remote server. At the time of this writing, web popups are only available on device builds and will not work on the simulator.

The syntax for a web popup can be either:
native.showWebPopup(url [, options]) or native.showWebPopup(x, y, width, height, url [options]). Parameters include:

- url – the url of the local or remote web page. By default this is assumed to be an absolute URL (i.e. use the entire http address).
- x, y – left top corner of the popup.
- width, height – dimensions of the popup window.

Options – optional table/array parameters

- options.baseURL – if set, allows the use of relative URLs.
- options.hasBackground – Boolean that sets an opaque background if true (default is true).
- options.urlRequest – sets a listener function to intercept all urlRequest events for the popup. Listener must return true to keep the popup open (default return is false).
-

```
native.showWebPopup( "http://www.BurtonsMediaGroup.com" )
```

```
native.showWebPopup( 10, 10, 300, 300,
"http://www.BurtonsMediaGroup.com" )
```

To remove a web popup, use the method:

```
native.cancelWebPopup()
```

Web Popup Example

In this example, a webListener function is used to find a webpage that is stored in the system.DocumentsDirectory. If it is not found or an error occurs, the listener will return false, which will cause the web popup to close.

```
local function webListener( event )
        local shouldLoad = true
        local url = event.url
        if 1 == string.find( url, "corona:close" ) then
            -- Close the web popup
            shouldLoad = false
        end

        if event.errorCode then
            -- Error loading page
            print( "Error: " .. tostring( event.errorMessage )
            shouldLoad = false
        end

        return shouldLoad
end

local options = { hasBackground=false,
baseUrl=system.DocumentsDirectory, urlRequest=webListener }
native.showWebPopup( "localpage1.html", options )
```

Summary

Please do note that widgets are still considered in beta at the time of this writing, so some functionality may change. If something doesn't seem to be working correctly visit the Ansca Mobile website and check for changes to the Widget API.

Assignments

1. Create an app that displays a number between 1 and 100 that updates as the user moves a slider.

2. Rewrite Project 13.1 using the table widget.

3. Using WebPopup widget, load the jpg image as was demonstrated in Project 14a.

4. Advanced: Create your own theme based upon the themes_ios.lua that are available from the Ansca Mobile website or included in the project downloads. Design your own colors, look, and feel to your widgets.

5. Using the newTabBar widget, create an app that allows you to change pages and see what will be served for breakfast, lunch, and dinner (each meal should be a static page). Challenge: If a meal menu is available online, pull the data from the Internet and load it into the appropriate page.

Chapter 16:
Rotten Apple - a Tower Defense Game

In this final instructional chapter we will create the first two levels of a tower defense game called Rotten Apple. I consider this a final beta version of the game. For the version that was submitted to the iTunes and Amazon stores additional runner, throwers, and levels were added. In this chapter we will specifically examine how to:

- use sprites and tiles to create a game map
- map navigation
- handle triggers
- handle pause/resume events
- save and load the game state for resuming a game later
- handle multiple game levels

Rotten Apples – Inspiration and Resources

This game was inspired by many mis-spent summers growing up in rural Indiana where we had an apple tree that produced very small fuit. While unpleasant to eat, the apples made great ammunition to be used on siblings and friends alike. In this tower defense style game, the goal is to position towers (in this case, individuals with a supply of apples) to defend your clubhouse from the competing gang who want to take over the clubhouse.

Graphics and levels were created by Brandon Burton of http://www.GeeklyEntertainment.com and are available in the resource download. The full version of the game is available on iTunes and the Google app store.

I decided to take a different approach in working with sprites for this project rather than using the API methods previously used for sprites. We will be using three tools: Tiled for creating the levels, Lime to handle the level loading, and SpriteLoq for handling the sprite sheet animations.

To simplify the development process, we have pre-created two levels using Tiled (http://mapeditor.org). The sprites were created in Flash and converted into spritesheets using Spriteloq (http://loqheart.com) which is also used for loading and animating the sprites. Finally, to help with loading and handling the map, I am using a lite version of Graham Ransom's Lime (yes, that's right, lime lite). You can get the full version of Lime at http://www.justaddli.me, which includes parallax and many additional features. Additional demonstration files are included in the resource folder under LimeLite.

Note: This is not the only way to build a tower defense game. There are many approaches that could be used. There are very few things in life that can only be accomplished one way.

Run from anyone who tells you different; they are either a fool, a cult leader, or want to sell you something you don't need.

I am targeting the tablet market for this game (specifically iPad and Kindle Fire). Thus I have configured the config.lua file accordingly. If you want to make your game to run on a smaller resolution device, you can adjust your resolution accordingly. I found, after some experimentation, that the 800 x 600 resolution worked very well for this particular map.

config.lua

```
application =
{
      content =
      {
            width = 600,
            height = 800,
            scale="letterbox",
            fps = 30,
            antialias = false,

            imageSuffix =
            {
                  ["@2"] = 2
            }
      }
}
```

The game will only support landscape, so I have configured the build.settings file in this way:

build.settings

```
settings =
{
      orientation =
      {
            default ="landscapeRight",
            supported =
            {
                  "landscapeLeft","landscapeRight"
            },
      },
}
```

Adding Sprite Animations

Rather than spend many pages on the art pipeline, let me quickly review what has been done and how it was accomplished. First, I needed sprites for the game. These were all created in Flash and exported as individual flash swf files. The Flash swf files were loaded into SpriteLoq as different sprite sheets. I created a sprite sheet for the runners (the sprites trying to reach the clubhouse), the throwers (the 'towers' defending the clubhouse),

the sprite sheet with level background information to be used by Tiled, and the clubhouse sprite. In all, a total of four sprite sheets saved as png files.

To ensure that my sprites were loading correctly, I started my main.lua file by just loading the sprite sheet. First I hide the status bar and load the loqsprite external library (which is located in the folder with the main.lua file). Then I create a new factory (a factory is the term used by loq sprite to refer to a group of sprite animations) for all of my runner sprites.

main.lua

```
display.setStatusBar(display.HiddenStatusBar)
local loqsprite = require("loq_sprite")
local runnerFactory = loqsprite.newFactory("runnersheet")
```

I assign the animation group called runner three (which is the red runner running up the screen) and set the start location at the bottom of the screen, 100 pixels in from the edge.

```
local runner1 = runnerFactory:newSpriteGroup("red runner up")
runner1.x = 100
runner1.y = display.contentHeight
```

Using the play() command, I start the animation sequence and use transition.to to move the runner from the bottom of the screen to the top.

```
runner1:play()
transition.to(runner1, {y =40, time = 5000})
```

If you want to see all of the available animation sequences available in your terminal window, you can use the atrace command supplied by loqsprite:

```
atrace(xinspect(runner1:getSpriteNames()))
```

```
= > atrace (00:00:00:132) table: 0x18ab650 {
 [1] => "rotten runnertwo"
 [2] => "greenRunnerOne"
 [3] => "rotten runner three"
 [4] => "runner down star"
 [5] => "green runner down star"
 [6] => "runner three"
 [7] => "green runner four"
 [8] => "runner four"
 [9] => "green runner three"
 [10] => "rotten runner one"
 [11] => "rotten runner four"
 [12] => "green runner two"
 [13] => "rotten runner down star"
 [14] => "runner one"
 [15] => "runner two"
```

Partial atrace list for spritesheet

I Need a Map!

Now that we know that we can add the sprite, let's move to the next phase and load our map that was created in Tiled. Using a lite version of Lime (included in the resource files), it only takes three lines of code to load our map that was created in Tiled. The Lime Lite external library files are placed in the same folder as our main.lua file. I have bolded the new lines of code:

```
display.setStatusBar(display.HiddenStatusBar)
local loqsprite = require("loq_sprite")
local runnerFactory = loqsprite.newFactory("runnersheet")
local lime = require("lime")
```

Using the lime load map function, we can load the level map created in Tiled.

```
-- Load and build map
local map = lime.loadMap("Lvl1.tmx")
```

Finally, we can display the map:

```
local visual = lime.createVisual(map)

local runner1 = runnerFactory:newSpriteGroup("red runner up")
runner1.x = start.x
runner1.y = start.y

runner1:play()
transition.to(runner1, {y = 40, time = 5000})
atrace(xinspect(runner1:getSpriteNames()))
```

One note on using Tiled with the lite version of Lime: You need to save the map as a straight XML file (set in the preferences menu). Do not use the 64base setting with Lime Lite; it will not load.
Now we have an animated sprite running over the top of the map. Next, let's work on getting the runner to run along a path.

Two Roads Diverged

Helping your sprites find their way through a map is of critical importance in many types of games. There are two primary methods used for path finding: A* and waypoints.

A* path finding is great for any environment where the path changes due to new elements or a path might become blocked. I have seen many tower defense games use this method very successfully. It is efficient, but also over-kill for our game.

Waypoints rely upon pre-defined points on the map that the character must pass through. Waypoints don't have as much overhead as A*, which must continuously calculate the best path to the goal. With waypoints we just need to define the turn locations on our map. Fortunately this is easily done in Tiled by adding an objects layer. Tiled saves one file when it creates a level: a tmx file, which is just an XML file describing the location of the level. It does require the associacted sprite sheet that was used to build the level to be in the same folder. For our game, I have added WP objects as waypoints on both maps:

Objects can be imported by Lime. Then, we just find the next waypoint and use a transition.to to move our sprite toward the goal.

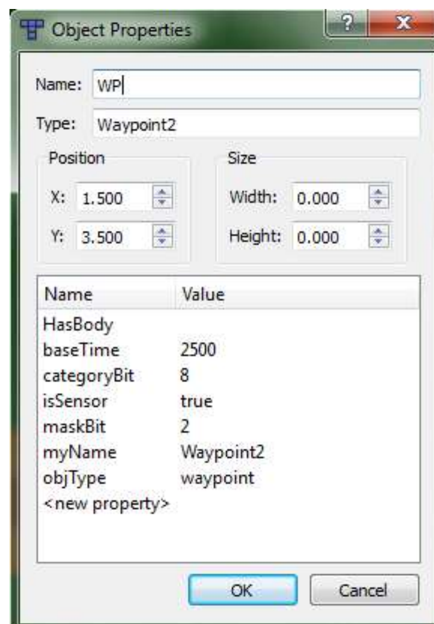
Space, The Final Frontier

One of the challenges with any game, whether it is 2D or 3D, is dealing with world space verses screen space. Most games use some type of coordinate system that must then be mapped to screen locations so that tapping and dragging have some type of meaning. Understand that while our maps are contained to be viewable all at once, it is possible to create much larger maps that can be scrolled and moved. Our world map is only limited by our imagination (and the amount of RAM on the device).

If our map was larger than one screen, we would need to translate the locations to the world space of the map. That's where Lime comes in. Lime includes several tools that we can use to convert a screen location into a world location and vice versa. It also allows for the world to be centered on the player or the world location to be changed by dragging. Fortunately, we don't have to worry about any of those issues for this game.

Rat Race

It's time to get our runner moving through the map. As I was designing this game, I put a lot of thought into how I wanted to handle each of the levels. I wanted to be able to easily add levels while keeping level specific coding to a minimum. The waypoints that I previously mentioned allow me to set the path through the map without having level-specific coding. By creating these objects in Tiled, I was able to set it so that the waypoints would be a physics body (the HasBody property), how long in miliseconds it should take to get to the next waypoint (baseTime), that it is a physics senor (which means collision events can occur, but no other physics reaction happens), and the name of the waypoint to make tracking easier in the code.



Through trial and error, I found that all waypoints at the end of an upward run had to be placed a tile higher due to collisions occurring with the sprites head. If I left the waypoint on the path, the collision occurred too early and the sprite would leave the path.

Let us assume that we are starting fresh on the main.lua file (this version is saved as "main – Rat Race.lua" in the files). First we will hide the status bar, load the loq sprite, lime and physics. Then start the physics engine and set the gravity to 0 (i.e. no gravity in this game, we don't want our sprites to fall off the bottom of the screen). Physics is just being used to handle collisions for this game. Last thing in this first section is to load our sprite sheet through loqsprite.

main.lua

```
display.setStatusBar(display.HiddenStatusBar)
local loqsprite = require("loq_sprite")
local lime = require("lime")
local physics = require("physics")
physics.start()
physics.setGravity(0,0)
local runnerFactory = loqsprite.newFactory("runnersheet")
```

Next, we need a couple of arrays to keep track of elements in the game. In the previous example I used runner1 to track the sprite. Since we will have multiple runners in the very near future, we should go ahead and set this to an array. We also need an array to track the waypoints that are loaded from the map.

```
-- A few variables to keep track of game elements
local runner = {}      -- array for tracking runners
local waypoint = {}    -- array for tracking waypoints
```

The pathfinder function handles all of the movement of the runner along the path. Pathfinder is called as a collision event. While we will need to modify it when it is time to handle runners getting hit by apples, for now it just handles moving the runner sprites from waypoint to waypoint.

To begin with, we will create a couple of local variables as arrays to hold the runner information and the waypoint information. Notice that I do have a print command at the beginning of the function so that I know whenever the function is called. This was critical in the design phase of this routine.

Initially, the runners were progressing through the waypoints too quickly due to multiple collisions between the runner and the waypoint. This took forever to resolve. We will obviously comment this out later, but for now, leave it in to help with any troubleshooting you might have to do.

```
local function pathfinder(event)
     print("Pathfinder was called")
      local myObj = {}
```

```
local WPHit = {}
```

Now we will set the local variables we just declared to their respective objects from the collision event. I did this primarily to simplify the programming in the next section, but I am also using it to weedout any strange collisions that do not involve a runner (this will be important in the next phase of the app). By placing a return command after the else statement, I am telling the function to disregard any collisions that do not include a runner as one of the objects. Later we will add the possibility of an apple hitting the runner, but for now we can safely assume that the other object in the collision is a waypoint.

```
if(event.object1.myName=="Runner") then
     myObj = event.object1
     WPHit = event.object2
elseif (event.object2.myName=="Runner") then
     myObj = event.object2
     WPHit = event.object1
else
     return
end
```

Remember that I said the runners were experiencing multiple collisions with the same waypoint and it was causing them to run in strange directions? (Trust me, I did). To solve that problem I store information with the runner as to its next waypoint. If this isn't equal to the waypoint that a collision just occurred with, we tell the function to return and go about its business. If the next waypoint is what we collide with, then the function will check for the need to change direction of the runner sprite.

```
local wpgoal = "Waypoint"..myObj.nextWP
if(wpgoal ~= WPHit.myName) then
   return
end
```

This is the meat of the function. First we will check to see if we are at the last waypoint (if we are, then the runner made it to the clubhouse, which is bad for the player… but we will deal with that later). Once we know there is another waypoint to run to, we increment the nextWP.

Based upon the x and y of the waypoints, we can calculate which runner sprite should be loaded next. In the first instance, if the x of the current waypoint is less than the x of the next waypoint, then we need to load the runner that faces toward the right. Before we do that, we will cancel the transition.to so that the sprite transition stops trying to move toward the current waypoint, then load (or prepare) the animation sequence with spriteloq, play the animation, then set a new transition to the new waypoint. Notice that all of the needed information is stored in the waypoint objects. This will allow us to make quick adjustments to the game and add multiple levels with little additional programming.

I did find it necessary to tweek the x and y coordinates by subtracting 16 pixels from the end location. This ended up being much easier than trying to get the sprite to follow the path by changing the waypoints by 16 pixels.

```
      if (myObj.nextWP+1 ~= nil) then
         myObj.nextWP = myObj.nextWP+1
         if (waypoint[myObj.nextWP-1].x < waypoint[myObj.nextWP].x)
then
           print("Load Runner right")
           transition.cancel(myObj)
           myObj:prepare("red runner right")
           myObj:play()
           transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
         elseif (waypoint[myObj.nextWP-1].x > waypoint[myObj.nextWP].x)
then
           print("Load Runner left")
           transition.cancel(myObj)
           myObj:prepare("red runner left")
           myObj:play()
           transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
         elseif (waypoint[myObj.nextWP-1].y <
waypoint[myObj.nextWP].y) then
           print("Load Runner down")
           transition.cancel(myObj)
           myObj:prepare("red runner down")
      myObj:play()
      transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
        elseif (waypoint[myObj.nextWP-1].y > waypoint[myObj.nextWP].y)
then
      print("Load Runner up")
      transition.cancel(myObj)
      myObj:prepare("red runner up")
      myObj:play()
      transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
        end
    else
           -- clubhouse is goal
  end
end
```

This next section should look familiar. We load the map using lime and create the visual map. The new step is using the buildPhysical command. This pulls in the objects and creates them as isSensor for the physics engine.

```
-- Load and build map
local map = lime.loadMap("Lvl1.tmx")
local visual = lime.createVisual(map)
local physical = lime.buildPhysical(map)
```

Now that our physical map is built, it would be nice to know where those waypoints are located. By using the lime library, we can pull in all the objects that have the name WP (which I used for all the waypoints on the maps).

Using a for loop that counts from 1 to the number of waypoints that were imported, I set the waypoint array to equal the name of the waypoint I want stored at that location.

Then I use a nested for loop to step through all of the tiles and assign the tile object properties to the correct waypoint. Piece of cake!

```
local tilesObj= map:getObjectsWithName("WP")
for i = 1, #tilesObj, 1 do
      waypoint[i]="Waypoint"..i
end
for i = 1, #tilesObj, 1 do
      for j = 1, #tilesObj, 1 do
            if tilesObj[j].type== waypoint[i] then
                  waypoint[i]={}
                  waypoint[i].x,waypoint[i].y =
tilesObj[j]:getPosition()
                  waypoint[i].baseTime=tilesObj[j].baseTime
                 waypoint[i].myName=tilesObj[j].myName
                 print("Waypoint "..i..": x"..waypoint[i].x.." y
"..waypoint[i].y)
            end
      end
end
```

We are about to unleash the hounds (or runners in this case). In our runner array, we will create the first runner for testing purposes. After assigning it to the red runner up sprite animation, we set the starting x and y location, set a speed for the runner (which we aren't using yet, just planning ahead), set the goal waypoint (in nextWP), hitpoints (again, planning ahead), a generic name so that we know it is a runner when a collision occurs, and finally, we begin to play the animation.
Next, we will add the runner as a physics body so that collisions can occur and finally give the initial transition.to command.

```
--Create Runner
runner[1] = runnerFactory:newSpriteGroup("red runner up")
runner[1].x=waypoint[1].x-16
runner[1].y=waypoint[1].y-32
runner[1].speed = 1
```

```
runner[1].nextWP = 2
runner[1].hp = 3
runner[1].myName="Runner"
runner[1]:play()

physics.addBody(runner[1])
transition.to(runner[1], {x= waypoint[2].x-16, y = waypoint[2].y-16,
time = waypoint[1].baseTime})

--atrace(xinspect(runner[1]:getSpriteNames()))

local update = function( event )
     map:update( event )
end
```

Now we just need to add our event listeners and we are ready to see our runner follow the path.

```
Runtime:addEventListener( "enterFrame", update )
Runtime:addEventListener("collision", pathfinder)
```

On Your Mark…

Now let's add a few additional runners to our game. In this version of the game I plan for three different types of runners: a Green Runner, a Red Runner, and a Brown or Rotten Runner.

The stats for each runner are (at least at design time):

Green Runner
- hp (hitpoints): 3
- speed: 1

Red Runner
- hp: 5
- speed: 0.8   -- The smaller the number, the faster they run.

Rotten Runner
- hp: 10
- speed: 1.1

It is my design that we will have 3 types of towers/apple throwers: green, red, and rotten. The green apple thrower will do 1 point of damage with each hit, the red apple thrower will do 2, and the rotten apple thrower will do 3 points of damage with the possibility of area

damage. This should make for a fun tower defense game. Of course, these numbers are all subject to change once we start play-testing the app.

It is time to add the routines to create the runners. At this point I am not worrying about differing levels of difficulty. I want to randomly generate 10 of the 3 types of runners to the current level.

First, we need to add a new variable at the top of main.lua to track how many runners have been added. Just add it with the other variables:

```
local runnerCount=0
```

Next, we need to make a few changes to pathfinder. I have bolded all of the additions and changes. These little changes allow pathfinder to be able to change the sprite for any of our three sprite sets.

```
local function pathfinder(event)
    print("Pathfinder was called")
      local myObj = {}
      local WPHit = {}
      if(event.object1.objType=="Runner") then
            myObj = event.object1
            WPHit = event.object2
      elseif (event.object2.objType =="Runner") then
            myObj = event.object2
            WPHit = event.object1
      else
            return
      end
     local wpgoal = "Waypoint"..myObj.nextWP
     if(wpgoal ~= WPHit.myName) then
           return
     end

     if (myObj.nextWP+1 ~= nil) then
      myObj.nextWP = myObj.nextWP+1
      if (waypoint[myObj.nextWP-1].x < waypoint[myObj.nextWP].x) then
            print("Load Runner right")
            transition.cancel(myObj)
            myObj:prepare(myObj.myName.."right")
            myObj:play()
            transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
      elseif (waypoint[myObj.nextWP-1].x > waypoint[myObj.nextWP].x)
then
             print("Load Runner left")
             transition.cancel(myObj)
             myObj:prepare(myObj.myName.."left")
             myObj:play()
```

```
            transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
            elseif (waypoint[myObj.nextWP-1].y <
waypoint[myObj.nextWP].y) then
                      print("Load Runner down")
                     transition.cancel(myObj)
                     myObj:prepare(myObj.type.."down")
           myObj:play()
           transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
      elseif (waypoint[myObj.nextWP-1].y > waypoint[myObj.nextWP].y)
then
            print("Load Runner up")
            transition.cancel(myObj)
           myObj:prepare(myObj.myName.."up")
           myObj:play()
           transition.to(myObj, {x= waypoint[myObj.nextWP].x-16, y =
waypoint[myObj.nextWP].y-16, time = waypoint[myObj.nextWP-
1].baseTime})
     end
         else
            print("Reached the clubhouse!")
        end
end
```

Next, we will add the addRunner function. This function can be added just after the pathfinder function in main.lua. This function starts by incrementing runnerCount. To decide which type of runner, we will use a random function to return a number between 1 and 3. Once we know which type of runner to add, we can load the sprite group, set the speed of the runner (which doesn't do anything yet), the hitpoints and the type of runner. Type is used back in our pathfinder routine to generate the right sprite sequence of up, down, left, or right.

```
local function addRunner ()
     runnerCount=runnerCount+1
     local whichRunner = math.random(3)
     if(whichRunner == 1) then
     runner[runnerCount] = runnerFactory:newSpriteGroup("green runner
up")
      runner[runnerCount].speed = 1
      runner[runnerCount].hp = 3
      runner[runnerCount].myName ="green runner "
     elseif(whichRunner==2) then
     runner[runnerCount] = runnerFactory:newSpriteGroup("red runner
up")
      runner[runnerCount].speed = .8
      runner[runnerCount].hp = 5
      runner[runnerCount].myName ="red runner "
     elseif(whichRunner==3) then
```

```
        runner[runnerCount] = runnerFactory:newSpriteGroup("rotten
runner up")
        runner[runnerCount].speed = 1.1
        runner[runnerCount].hp = 10
        runner[runnerCount].myName="rotten runner "
      end
      runner[runnerCount].x=waypoint[1].x-16
      runner[runnerCount].y=waypoint[1].y-16
      runner[runnerCount].nextWP = 1
      runner[runnerCount].objType="Runner"
      runner[runnerCount]:play()
      physics.addBody(runner[runnerCount])
end
```

You might be wondering why we don't need to create a two-dimensional array with our runner array (i.e. issue the command runner[runnerCount]={}. One of the nice things accomplished with the spriteloq library call is to load the sprite image and create runner[runnerCount] as a two-dimensional array.

The last step is to replace all of our previous code defining the runner with these calls to addRunner:

```
addRunner()
timer.performWithDelay(2000, addRunner)
timer.performWithDelay(4000, addRunner)
```

I just placed 3 calls, spaced 2 seconds apart so that we can see if the addRunner function is working correctly.

Reducing Collisions

I didn't cover this in Chapter 8 because, well, we didn't really need it at that time and this is a fairly advanced topic. As I was working through this section of the chapter, I had the issue of things colliding that I didn't want to collide: Apples with Apples, Runners with Runners, Apples with Towers, Apples with Waypoints; you get the picture. It was mass chaos: human sacrifice, dogs and cats living together… mass hysteria! (And yes, that was a movie reference).

To resolve this problem, we will use a filter in the declaration of the addBody to tell the engine what can and cannot collide. The filter has two properties: CategoryBit and MaskBit. CategoryBit is a unique, binary number used to identify the category of body. MaskBit is the sum of all categoryBits that can collide with that body.

To handle this, I recommend that you make a worksheet like I have below. I have listed each category of body and assigned it a unique binary number (remember, binary counting goes 0, 1, 2, 4, 8, 16, 32, etc). In the table below, I assigned Apple the categoryBit of 1. The

apple can ONLY collide with a runner which has a categoryBit of 2. So the maskBit of Apple is 2.

The runner body is a bit more complex. It must be able to collide with the apple (categoryBit of 1) and waypoints (categoryBit of 8). Thus the runner will have a maskBit of 9 (apple + waypoint) so that it will collide with each.

Collision Worksheet

		Apple	Runner	Tower	Waypoint	Sum
Apple	CategoryBit	1				
	MaskBit	No	Yes (2)	No	No	2
Runner	CategoryBit		2			
	MaskBit	Yes (1)	No	No	Yes (8)	9
Tower	CategoryBit			4		
	MaskBit	No	No	No	No	0
Waypoint	CategoryBit				8	
	MaskBit	No	Yes (2)	No	No	2

While the waypoint categoryBit and maskBit are included in the Tiled Properties, the rest must be set at the time you add the body to the physics engine:

```
physics.addBody(runner[runnerCount], {density = 10, bounce = 0, filter
= {categoryBits =2, maskBits =1}})
```

Take the Shot – Taking Care of Collisions

Its time to setup how the towers will work, throwing the apple, and the rest of the various types of collisions that can occur. I am listing the entire app from beginning to end from this point forward. The previous section of the chapter was to show how to load your map, sprites and handle the basic collisions. As before, I will bold the changes to our code.

main.lua

```
display.setStatusBar(display.HiddenStatusBar)
local ui = require("ui")
local loqsprite = require("loq_sprite")
local lime = require("lime")
local physics = require("physics")
physics.start()
physics.setGravity(0,0)
local runnerFactory = loqsprite.newFactory("runnersheet") -- load spriteLoq
sheet

-- A few variables to keep track of game elements
local hit = audio.loadSound("hit 1.wav")
local runner = {}          -- array for tracking runners
```

```
local waypoint = {}  -- array for tracking waypoints
local runnerCount=0   -- how many runners are on the screen?
local apple = {}      -- array for tracking thrown apples
local appleCount = 0 -- How many apples have been thrown?
local sound = true   -- Should sound effects play?
local paused = false -- Is the game paused?
local textScore      -- display for score
local textDamage     -- display for damage
local textWave       -- display for wave count
local textApples     -- display for apples collected
local sndImage       -- image to turn sound on/off
local pauseImage   -- image to pause game
local splash          -- splash screen image
local newGameButton
local loadGameButton
local tower = {}          -- array for tracking towers
local towerTiles = {}    -- array for tracking map tiles and where
towers are set.
local towerCount=0       -- number of towers deployed
local level = 1           -- current level
local levelStarted = false  -- to control game loop
local tick = 400      -- game loop speed
local map             -- stores the tiled map
local visual          -- stores viewable tiled map
local physical        -- holds the physical from tiled
local tilesObj         --holds the tiled objects from tiled
local score = 0       -- Game score
local applesCollected = 150  -- apples gathered
local damage = 0          -- damage done to clubhouse
local clubhouse = {}  -- store clubhouse data
local greentower      -- drag-n-drop tower
local redtower        -- drag-n-drop tower
local rottentower     -- drag-n-drop tower
local waveCount = 1  -- current wave
local wave ={}        -- array for waves 1=green, 2= red, 3 = rotten,
0 = random
      wave[1] = {1,1,1,1,1,1,1,1,1,1}
      wave[2] = {1,1,1,1,2,1,1,1,1,2}
      wave[3] = {1,2,2,2,2,3,2,1,1,2}
      wave[4] = {2,2,3,1,2,2,3,2,3,3}
      wave[5] = {3,2,1,3,2,1,3,2,1,3}
      wave[6] = {3,2,2,1,3,2,2,1,0,0}
      wave[7] = {0,0,0,0,0,0,0,0,0,0}
      wave[8] = {3,0,3,0,3,0,3,3,2,0,3,1,2,2,2}
      wave[9]= {3,0,0,3,0,3,0,3,3,3,0,0,0,0,3,3,0,0,3,0}
      wave[10]={3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3}
```

Are We There Yet? – adding the clubhouse

The first function in our app handles the clubhouse sprite. As runners reach the clubhouse it is progressively damaged.

```
local function clubhouseDamage()
      --  print("Reached the clubhouse!")
        damage = damage + 1
        if(damage < 3) then
            clubhouse:prepare("club house75")
        elseif(damage < 6) then
            clubhouse:prepare("club house50")
        elseif(damage < 8) then
            clubhouse:prepare("club house25")
        elseif(damage >= 10) then
            clubhouse:prepare("club house00")
            print("Game Over!")
        end

end

local function pathfinder(event)
      --print("Pathfinder was called")
      local myObj = {}  -- myObj will always be the runner
      local WPHit = {}
      if(event.object1.objType=="Runner") then
            myObj = event.object1
            WPHit = event.object2
      elseif (event.object2.objType=="Runner") then
            myObj = event.object2
            WPHit = event.object1
      else
            return  -- collision was not with a runner
      end
-- To help with troubleshooting print object types:
print("My Name 1: "..myObj.myName..", "..myObj.objType)
print("My Name 2: "..WPHit.myName..", "..WPHit.objType)

    -- Check if Runner collision is a waypoint;
      --handle change in direction if it is.
```

I added the first if statement so that only waypoint/runner collisions will be handled. The first thing we will check for now is if it is the “Finish” or clubhouse waypoint. If it is, then we will remove the runner and damage the clubhouse. Next, we check if we are at the right goal, if we are not, then the collision is disregarded. Instead of ending there, we will use an elseif to continue the evaluation since it is more processing efficient.

```
    if(WPHit.objType == "waypoint") then
      if(WPHit.myName == "Finish" ) then
            myObj:removeSelf()
            myObj.myName = nil
            clubhouseDamage()
    else
     local wpgoal = "Waypoint"..myObj.nextWP
        --Check to make sure it is the next waypoint

        if(wpgoal ~= WPHit.myName) then
```

```
            return     --hit the same waypoint again
          elseif (myObj.nextWP+1 ~= nil) then
            myObj.nextWP = myObj.nextWP+1
```

If we have reached this point in the if..then structure, then we know that we need to go to a new waypoint. The next line of code handles the transition to the new object. The transition is stored in myObj.currentTransition. By storing it in the runner data (remember myObj = the runner that had a collision event), we will be able to pause the transition later in the app. The transition determines the next waypoint location by passing the next waypoint number that is stored in myObj. I found that subtracting 16 from the x and y kept the runners in a straighter line as they were running up or down the screen.

The next change is a minor addition to handling the transition.to time sequence. By adding *myObj.speed to the end of the base time, we can speed up or slow down the runners. At this time, the red runner will be 20% faster and the rotten runner 10% slower than the green runner.

Finally, we will store how long it is expected to take the runner to reach the waypoint as runner.baseTime. This will allow us to resume the transition after a pause and keep the runner going at a reasonable speed (the alternative is a great deal of calculations to determine how far the next way point is). We will subtract from this baseTime in the game loop with each iteration of the game loop.

```
            myObj.currentTransition = transition.to(myObj, {x=
waypoint[myObj.nextWP].x-16, y = waypoint[myObj.nextWP].y-16, time =
(waypoint[myObj.nextWP-1].baseTime*myObj.speed)})
            myObj.baseTime=waypoint[myObj.nextWP-1].baseTime
```

With the transition and time calculations already done, we just need to determine which sprite to use. This simplifies our code and reduces repetition.

```
      if (waypoint[myObj.nextWP-1].x < waypoint[myObj.nextWP].x) then
            --print(myObj.myName.."right")
            myObj:prepare(myObj.myName.."right")
            myObj:play()
       elseif (waypoint[myObj.nextWP-1].x > waypoint[myObj.nextWP].x)
then
            --print(myObj.myName.."left")
            myObj:prepare(myObj.myName.."left")
            myObj:play()
      elseif (waypoint[myObj.nextWP-1].y < waypoint[myObj.nextWP].y)
then
            --print(myObj.myName.."down")
            myObj:prepare(myObj.myName.."down")
            myObj:play()
      elseif (waypoint[myObj.nextWP-1].y > waypoint[myObj.nextWP].y)
then
            --print(myObj.myName.."up")
            myObj:prepare(myObj.myName.."up")
            myObj:play()
```

```
      end
    end
  end
  return   -- no need to run the rest of the function
end
```

We are still in our Pathfinder function. Pathfinder has become the catch-all for collisions of all types. Moving on to the next type of collision: runner and tower. The first thing we will do is check to make sure the WPHit object is a tower. If it is, is the tower ready to throw another apple? The timeTilThrow will eventually be managed by the game loop, but since that hasn't been implemented yet; we will give it a free first throw in the tower configuration below.

```
    --Check if the runner collision is with tower
    if(WPHit.objType == "tower") then
      if( WPHit.timeTilThrow <= 0) then      --can throw again
```

To manage the throw, we will first reset the count for being able to throw again by setting the timeTilThrow property equal to throwSpeed. This allows each tower type to have its own speed. Next, we will make the apple visible by setting the alpha to 1. We will also set the active property to true so that we know that the apple is in use (I'll explain why shortly). Finally, using the setLinearVelocity physics method, we determine where to throw the apple by subtracting the towers location from the runner location.

```
        WPHit.timeTilThrow = WPHit.throwSpeed
        apple[WPHit.number][tower[WPHit.number].appleCount].alpha=1
        apple[WPHit.number][tower[WPHit.number].appleCount].active = 
true
apple[WPHit.number][tower[WPHit.number].appleCount]:setLinearVelocity(
(myObj.x - WPHit.x),(myObj.y-WPHit.y))
           WPHit.needApple = true
        end
        return  -- no need to run the rest of the function
    end
```

For the final part of the pathfinder function, we will handle the apple/runner collisions. First, we will ensure that the WPHit object was an apple. If it is, then we will subtract points from the runner's hitpoints and remove the apple by setting the alpha to zero.

It is necessary to keep track of apples that are in use. This is due to the issue that the apples have to be added as physics bodies prior to any collisions occurring. To manage that, I created an apple for each tower. Once the apple is thrown, a new apple will be created for the tower in the game loop routine. To help keep track of apples that have been thrown, we will set the timeToLive for each apple thrown to 3.5 seconds, thus keeping the screen from becoming over populated with thrown apples.

```
      -- handle apple/runner collisions
      if(WPHit.objType=="apple") then
         myObj.hp= myObj.hp-WPHit.damage
         --print(WPHit.myName)
         if(WPHit~=nil) then       --make sure we don't hit have
multiple hits
           WPHit:removeSelf()  --remove the apple when it hits
           WPHit=nil
         end
```

How about some sound effects? If sound hasn't been turned off (i.e. sound is true) then play the hit sound.

```
      if sound then
           audio.play(hit)
      end
```

To wrap up the pathfinder/apple collision portion of this function, we will first check to see if the runner has already been knocked out (avoiding multiple collisions and attempts to delete the same object twice) and play the stars sprite animation.

The stars sprite animation is set to fade out over 2 seconds. Then we will add to the applesCollected and score. Finally, we can't remove an object that is part of a collision until after the collision is done, so we will instead set the name and objType to "down" as in knocked down, so that we know to remove them later in the game loop.

```
      if (myObj.hp <= 0 and myObj.myName ~= nil) then
           --    print("Do star sequence")
           transition.cancel(myObj.currentTransition)
           myObj:prepare(myObj.myName.."stars")
           myObj:play()
           transition.to(myObj, {time = 2000, alpha = 0})
           applesCollected = applesCollected + myObj.appleValue
           score = score + myObj.score
           myObj.myName = "down"
           myObj.objType="down"
       end
      end
end
```

For the addRunner function, we are just going to add two if..then statements and some additional variables for the runner. The first will handle the levelStarted variable. We don't want the game loop running until after the runners are on the screen. Once runners are on the screen, the level has officially begun.

The second if..then statement is there to handle random runners. If a "0" is specificed, we will randomly pick a runner to add to the wave.
We will also add the apple value and score value for each runner.

```
local function addRunner ()
   if levelStarted == false then
     levelStarted = true
   end
   if(whichRunner==0) then
     whichRunner= math.random(1,3)
   end
   runnerCount=runnerCount+1
    if(whichRunner == 1) then
       runner[runnerCount]= runnerFactory:newSpriteGroup("green runner
up")
       runner[runnerCount].speed = 1
       runner[runnerCount].hp = 3
       runner[runnerCount].myName="green runner "
         runner[runnerCount].appleValue = 1
         runner[runnerCount].score = 3
    elseif(whichRunner==2) then
       runner[runnerCount] = runnerFactory:newSpriteGroup("red runner
up")
       runner[runnerCount].speed = .8
       runner[runnerCount].hp = 5
       runner[runnerCount].myName="red runner "
         runner[runnerCount].appleValue = 3
         runner[runnerCount].score = 5
    elseif(whichRunner==3) then
       runner[runnerCount] = runnerFactory:newSpriteGroup("rotten
runner up")
       runner[runnerCount].speed = 1.1
       runner[runnerCount].hp = 10
       runner[runnerCount].myName="rotten runner "
         runner[runnerCount].appleValue = 5
         runner[runnerCount].score = 10
    end
    runner[runnerCount].x=waypoint[1].x-16
    runner[runnerCount].y=waypoint[1].y-16
    runner[runnerCount].nextWP = 1
    runner[runnerCount].objType="Runner"
         runner[runnerCount].baseTime=0  --placeholder to resume
         runner[runnerCount].currentTransition = nil   -- placeholder
so that transition can be canceled or paused
    runner[runnerCount]:play()
```

The only addition to the addRunner function is to add a few physics properties, including the categoryBits and maskBits. This ensures that the runner only collides with the apples and waypoints.

```
    physics.addBody(runner[runnerCount], {density = 10, bounce = 0,
filter = {categoryBits =2, maskBits =1}})
end
```

The orchard function takes care of making sure an apple is available for the tower/thrower.

It is necessary that we know how many apples have been thrown by tower, so that they can all be deleted from memory between levels. Sine an apple wil no longer be needed by a tower, we will set the needApple to false.

```
local function orchard(towerCounter)
      local i= tower[towerCounter].appleCount + 1
      tower[towerCounter].attackArea.needApple = false
      tower[towerCounter].appleCount =i
```

Now we can setup the apple to be thrown. First, we will determine the apple type (green, red, or rotten). Then, after loading the correct sprite, we can tell the animation to play (which gives the apple its spin).

```
      apple[towerCounter] = {}
      if(tower[towerCounter].attackArea.myName =="green tower") then
            apple[towerCounter][i]=runnerFactory:newSpriteGroup("apple
Green")  -- load the green apple animation
            apple[towerCounter][i].myName = "green apple"
      elseif(tower[towerCounter].attackArea.myName =="red tower") then
            apple[towerCounter][i]=runnerFactory:newSpriteGroup("apple
Red")  -- load the red apple animation
            apple[towerCounter][i].myName = "red apple"
      elseif(tower[towerCounter].attackArea.myName =="rotten tower")
then
            apple[towerCounter][i]=runnerFactory:newSpriteGroup("apple
Rotten")  -- load the rotten apple animation
            apple[towerCounter][i].myName = "rotten apple"
      end
```

Next, we will set how much damage the apple will do based upon its type and the time to live. We will make the apple invisible (alpha = 0) until it is needed and let the routine know that it is not active.

```
      apple[towerCounter][i]:play()
      apple[towerCounter][i].damage = tower[
towerCounter].attackArea.damage
      apple[towerCounter][i].objType = "apple"
      apple[towerCounter][i].timeToLive = 3500
      apple[towerCounter][i].isBullet = true
      apple[towerCounter][i].alpha = 0
      apple[towerCounter][i].active = false
```

Finally, we set the x and y location of the apple based upon the tower location, and add the apple as a physics body that will behave like a bullet (i.e., constantly checked for collisions). The categoryBits and maskBits are set so that it will only collide with runners.

```
      apple[towerCounter][i].x, apple[towerCounter][i].y =
tower[towerCounter].x, tower[towerCounter].y
      physics.addBody(apple[towerCounter][i], {density = .01, bounce =
0, filter={categoryBits=1, maskBits=2}})
```

```
end
```

Another new addition in this section is the tower itself. To manage tower addition, we will create an addTower function. This will be called each time the player drags and drops a thrower/tower on to the screen. To handle collisions, we will add a circle that has an 80 pixel diameter. I have set the circle color to white and partially faded so that it is visible on the screen. The final line of this function calls the orchard function above to create the first apple to be used by this tower.

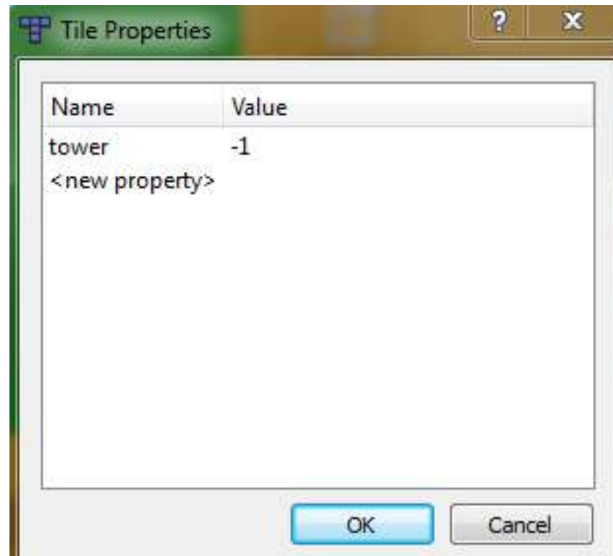
```
local addTower = function(towerType, towerx, towery)
	towerCount = towerCount+1
	if(towerType =="green") then
		tower[towerCount]=runnerFactory:newSpriteGroup("green apple thrower")
		tower[towerCount].range= 80		-- base range of tower in pixels
		tower[towerCount].x = towerx
		tower[towerCount].y = towery
		tower[towerCount].number=1
		tower[towerCount].attackArea = display.newCircle(tower[towerCount].x, tower[towerCount].y, tower[towerCount].range)
		tower[towerCount].attackArea.myName = "green tower"
		tower[towerCount].attackArea.damage =1	-- amount of damage
	elseif(towerType=="red") then
		tower[towerCount]=runnerFactory:newSpriteGroup("red apple thrower")
		tower[towerCount].range= 80		-- base range of tower in pixels
		tower[towerCount].x = towerx
		tower[towerCount].y = towery
		tower[towerCount].number=2
		tower[towerCount].attackArea = display.newCircle(tower[towerCount].x, tower[towerCount].y, tower[towerCount].range)
		tower[towerCount].attackArea.myName = "red tower"
		tower[towerCount].attackArea.damage =2	-- amount of damage
	elseif(towerType=="rotten") then
		tower[towerCount]=runnerFactory:newSpriteGroup("rotten apple thrower")
		tower[towerCount].range= 80		-- base range of tower in pixels
		tower[towerCount].x = towerx
		tower[towerCount].y = towery
		tower[towerCount].number=3
```

```
          tower[towerCount].attackArea =
display.newCircle(tower[towerCount].x, tower[towerCount].y,
tower[towerCount].range)
          tower[towerCount].attackArea.myName = "rotten tower"
          tower[towerCount].attackArea.damage =3     -- amount of
damage
     end

     tower[towerCount].appleCount=0
     tower[towerCount].attackArea:setFillColor(255,255,255,15)
     tower[towerCount].attackArea.objType = "tower"
     tower[towerCount].attackArea.number=towerCount
     physics.addBody(tower[towerCount].attackArea,
{filter={categoryBit=4, maskBit=0}})
     tower[towerCount].attackArea.isSensor=true
     tower[towerCount].attackArea.throwSpeed =1000      --
milliseconds between throws
     tower[towerCount].attackArea.timeTilThrow = 0        --
milliseconds until can throw again
     tower[towerCount].attackArea.needApple = true
     orchard(towerCount)              --create first apple for tower

end
```

We now have a functional tower, collision occurring, and our runner getting hit by an apple! Seems like a good place to add the ability more towers!

Adding Towers: Dragging Towers to the Screen

The startDrag function is a modified version of what we used in chapters 9 and 10 to move the starship. In this version, I removed the physics since the prototype towers are not physics bodies. Instead, I focused on modifying the last section to determine whether the tower can be set in a specific location. To manage the tower location selection, I found it necessary to add a property to the Tiled map to track if it was an acceptable location (i.e. a grass area and not the road, but on the tiled map).



By setting the tile properties (by right clicking on the spritesheet tile), I could set a new property called tower. I decided that a "-1" was an area where a tower or thrower could not be set. A "0" would be an available area. After a tower is set, it is changed to the value of the tower's number. This keeps towers from being set on top of each other. Note that all of the code in this section is new to our app.

```
local function startDrag( event )
     if paused then
          return
     end
     local t = event.target

     local phase = event.phase
     if "began" == phase then
          display.getCurrentStage():setFocus(t)
          t.isFocus = true
          --Store inital position
          t.x0 = event.x - t.x
          t.y0 = event.y - t.y
     elseif t.isFocus then
       if "moved" == phase then
          t.x = event.x - t.x0
          t.y = event.y - t.y0
       elseif "ended" == phase or "cancelled" == phase then
          t.isFocus = false
          display.getCurrentStage():setFocus(nil)
```

Once the drag and drop has ended (i.e. the phase has changed to "ended"), we will check to see if there are enough apples to make the purchase of the tower. If not, the tower will be returned back to the side.

```
          if(t.cost <= applesCollected) then
               applesCollected=applesCollected - t.cost
```

```
            else
                  t.x=t.initialX
                  t.y=t.initialY
                  return
            end
            local dropLocation = {x = t.x, y = t.y}
```

Next, we check the map tile location. If there is no tile (it equals nil), then the tower was not dropped on the map. Using a while loop, we can quickly check to see if the tower location is okay (the tower property equals 0). If it does equal 0, then we will create a new tower on that tower through the addTower function and return the prototype tower to the side.

```
            if(map:getTileAt(dropLocation) ~= nil) then
              local testTile = map:getTileAt(dropLocation)
              local found = false
              local i = 0
              while (not found) do
                  i = i + 1
                  if(towerTiles[i].row == testTile.row and
towerTiles[i].column == testTile.column and towerTiles[i].tower == 0)
then
                  -- good location
                  towerTiles[i].tower = t.myName
                  addTower(t.myName, t.x, t.y)
                  t.x=t.initialX
                  t.y=t.initialY
                  found = true
            end
            if (i == #towerTiles) then
            -- location wasn't found or was bad/inuse
                  t.x=t.initialX  --return tower to the side
                  t.y=t.initialY
                  found = true -- not really, need to break out of the
loop
            end
      end
      else
      -- location wasn't found or was bad/inuse
            t.x=t.initialX  -- return the tower to the side
            t.y=t.initialY
      end
     end
  end
  return true
end
```

Of course we need the prototype towers for dragging and dropping! This function creates the initial prototype towers.

```
local function initializeTowers()
            greentower= runnerFactory:newSpriteGroup("green apple
thrower")
            greentower.myName="green"
            greentower.x=display.contentWidth-100
            greentower.initialX=display.contentWidth-100
            greentower.y=150
            greentower.initialY=150
            greentower.objType= "prototower"
            greentower:addEventListener( "touch", startDrag )
            greentower.cost = 50
            redtower= runnerFactory:newSpriteGroup("red apple thrower")
            redtower.myName="red"
            redtower.x=display.contentWidth-100
            redtower.initialX=display.contentWidth-100
            redtower.y=250
            redtower.initialY=250
            redtower.objType= "prototower"
            redtower:addEventListener( "touch", startDrag )
            redtower.cost=75
            rottentower= runnerFactory:newSpriteGroup("rotten apple
thrower")
            rottentower.myName="rotten"
            rottentower.x=display.contentWidth-100
            rottentower.initialX=display.contentWidth-100
            rottentower.y=350
            rottentower.initialY=350
            rottentower.objType= "prototower"
            rottentower:addEventListener("touch", startDrag )
            rottentower.cost=100
end
```

What's the Score?

The next two functions handle creating and displaying the score, apples collected, wave number, and damage that has been done to the clubhouse. We've been doing this sort of thing since chapter 1, so we'll keep on moving!

```
-- Display lives and score
local function initializeStatus()
   textApples = display.newText("Apples: "..applesCollected,
display.contentWidth - 120, 30, nil, 12)
   textScore = display.newText("Score: "..score, display.contentWidth
- 120, 10, nil, 12)
   textDamage = display.newText("Damage: "..damage,
display.contentWidth -120, 50, nil,12)
   textWave = display.newText("Wave "..waveCount.." of 10",
display.contentWidth -120, 70,nil, 12)
   textApples:setTextColor(255,255,255)
   textScore:setTextColor(255,255,255)
```

```
   textDamage:setTextColor(255,255,255)
   textWave:setTextColor(255,255,255)
end


local function updateScore()
      textApples.text = "Apples: "..applesCollected
      textScore.text = "Score:  "..score
      textDamage.text="Damage: "..damage
      textWave.text="Wave "..waveCount.." of 10"
end
```

Let's Get this Game Going!

Time to add the game control features. First, we will add the startWave function. StartWave handles adding the runners to the screen one at a time with a variable amount of distance between each runner.

Closures

It was necessary to use a closure in this routine. A closure is a function that is fully contained within another function. It has full access to all variables in the initial function (because it is local to the function). It also allows us to pass a variable to something that doesn't like to have variables passed to it!
In our function startWave, we need to call timer.performWithDelay and pass it the runner information and the time delay. The problem is that if you pass a function to timer.performWithDelay with paramaters, it will run instantly instead of waiting. Try it for yourself: replace AR with addRunner(wave[wavenumber][i]). I'll wait…

Did you see what happened? ALL of the runners in that wave appear at the same time; no delay! This is obviously NOT a good thing! By using a closure, we can pass all the arguments, but they are not seen as parameters. Problem solved!

```
local function startWave(waveNumber)
  local AR
  local timerdelay = 0
  for i = 1, #wave[waveNumber] do
  -- use closure when using performWithDelay to send data
     AR = function() return addRunner(wave[waveNumber][i]) end
     timerdelay = timerdelay+ 500 + math.random(0,500)
     timer.performWithDelay(timerdelay, AR)
  end
end
```

Initialization

The initializeLevel function handles resetting and setting all the required variables as well as loading the map and tiles on to the screen. The first thing we will handle in this routine is resetting everything between levels. This will reduce memory leaks and improve performance as the game progresses.

```
local function initializeLevel()
      physics.stop()
      physics.start()
      physics.setGravity(0,0)
      if(map~= nil) then   --clean up everything if there was a
previous level
            map:destroy()
            for i = 1, runnerCount do
                  runner[i]:removeSelf()
            end
            for i=1,#tower do
                  for j=1, tower[i].appleCount do
                        if(apple[i][j] ~=nil) then
                              apple[i][j]:removeSelf()
                              apple[i][j]=nil
                        end
                  end
                  tower[i]:removeSelf()
                  tower[i]=nil
            end
            towerCount = 0
            runnerCount = 0
            towerCount = 0
            appleCount = 0
            tower = nil
            tower = {}
            towerTiles = nil
            towerTiles = {}
            apple = nil
            apple = {}
            runner = nil
            runner = {}
            waypoint= nil
            waypoint={}
            clubhouse = nil
            clubhouse = {}
      end
```

Next, we will load the map. A little has been added to the routine to handle loading the tile information so that we will know where towers can be set.

```
-- Load and build map
   local currentLevel = "Lvl"..level..".tmx"
   map = lime.loadMap(currentLevel)
```

```
   visual = lime.createVisual(map)
   physical = lime.buildPhysical(map)
   tilesObj= map:getObjectsWithName("WP")
   initializeTowers()
   for i = 1, #tilesObj, 1 do
     waypoint[i]="Waypoint"..i
   end
   for i = 1, #tilesObj, 1 do
     for j = 1, #tilesObj, 1 do
        if tilesObj[j].type== waypoint[i] then
          waypoint[i]={}
          waypoint[i].objType = tilesObj[j].objType
          waypoint[i].x,waypoint[i].y = tilesObj[j]:getPosition()
          waypoint[i].baseTime=tilesObj[j].baseTime
          waypoint[i].myName=tilesObj[j].myName
        end
     end
   end

--  Find where towers can be set
   local TTiles = map:getTilesWithProperty("tower")
   for i = 1,#TTiles do
     towerTiles[i]={}
     towerTiles[i].row = TTiles[i].row
     towerTiles[i].column = TTiles[i].column
     towerTiles[i].tower = TTiles[i].tower
   end
    --map:showDebugImages()  -- shows all tile objects
```

Now that we know where the waypoints are and what tiles can have towers, we can load the clubhouse on top of the final waypoint.

```
clubhouse=runnerFactory:newSpriteGroup("club house100")
 if(damage > 0 and damage < 3) then
     clubhouse:prepare("club house75")
  elseif(damage < 6) then
     clubhouse:prepare("club house50")
  elseif(damage < 8) then
     clubhouse:prepare("club house25")
  end
  clubhouse:play()
  for i = 1, #tilesObj, 1 do
     if(waypoint[i].myName=="Finish") then
          clubhouse.x=waypoint[i].x
          clubhouse.y=waypoint[i].y
     end
  end
```

We will load the mute sound image and pause image to the screen. The functions controlling the sound and pause are handled later in the app.

```
  sndImage = display.newImage("sound.png", display.contentWidth- 75, display.contentHeight- 75)

  pauseImage = display.newImage("pause.png", display.contentWidth-75, display.contentHeight - 130)
```

We have everything loaded, so time to begin the first wave of runners!

```
  -- release the hounds err... runners!
  startWave(1)

--  atrace(xinspect(runner[1]:getSpriteNames()))
end
```

Loop-De-Loop!

The gameLoop routine is called every 400 milliseconds by the system (as set by our tick variable back at the top of the app). The first thing we check for is if the game has been paused. Obviously there is no need to do a loop if the game is paused.

Now we do a run-through of the runners to see who has been knocked down. If they are down, then they are removed from the physics engine so that collisions cannot continue to occur with a down runner (it's not fair to hit a guy/girl when they are down). We will also handle subtracting from the baseTime for each active runner so that, should a pause occur, we can easily resume.

```
local function gameLoop()
     if paused then
        return
     end
     for i=1,runnerCount do
        if(runner[i].myName == "down") then
          physics.removeBody(runner[i])
          runner[i].myName =nil
        else
          runner[i].baseTime = runner[i].baseTime-tick
        end
     end
```

The next loop handles everything for towers between loops: Is the tower ready to throw again? Does the tower need an apple? Are their missed apples floating around?

First, we check if the timeTilThrow is greater than 0. If it is, then subtract the tick value, so that we can throw again. After that check, we will check on apple needs. If the tower needs an apple, then orchard is called. Finally, we check all the apples that this tower has thrown. If one of them missed, then check to see if its timeToLive value to see if it can be removed. Finally, any apples that have lived a full life are removed from the screen.

```
      -- handle tower timeTilThrow
      for i = 1, towerCount do
         if (tower[i].attackArea.timeTilThrow > 0) then
           tower[i].attackArea.timeTilThrow = 
tower[i].attackArea.timeTilThrow - tick
         end
         if(tower[i].attackArea.needApple == true) then
           orchard(i)
         end
         -- handle apples that missed
         for j = 1, tower[i].appleCount do
           if(apple[i][j] ~= nil) then
              if(apple[i][j].active == true) then
                apple[i][j].timeToLive = apple[i][j].timeToLive - tick
                if (apple[i][j].timeToLive <= 0) then
                    apple[i][j]:removeSelf()
                     apple[i][j]=nil
                end
              end
            end
         end
      end
```

Now that we have taken care of runners, towers, and apples; we can update the score and see if we are ready for a new level. After our score update, we will check to see if the level has started. If it has, then we need to see if all of the runners have been knocked down or made it to the clubhouse. We will assume that has happened, then check each runner to see if any of them are still active. If any of them are still active, we change waveFinished to false and return out of the function since all other processing for this function is to handle the change between waves and levels.

```
      updateScore()
      if(levelStarted == true) then
      -- ready for a new wave or level??
      waveFinished=true
      for i = 1, #runner do
         if (runner[i].myName ~= nil) then
           waveFinished = false
           return
         end
      end
```

Level and Wave Control

If you were wondering how to proceed to the next wave or level, this is it! This section (still in the gameLoop function) checks to see if the wave is finished. If it is and the current wave is 10, then it will proceed to level 2 (once we have more levels, we will change that code to level = level +1). We will give a bonus for completing the level – 1000 apples to spend on new towers! The waveCount is reset to 1 (starting the whole process over), and we will call the initializeLevel function to load the new map.

If the waveCount has reached 10 and we are on level 2, then the player has won the game. While we don't have a fancy "You Won" screen, we do print it in the terminal window (that's just as good, isn't it??).

If the player has just finished the wave, award them with 100 apples, increment the waveCount, reset the runnerCount and have the wave begin.

```
      --Wave & Level Control
         if(waveCount == 10 and waveFinished == true and level == 1)
then
            level = 2
            levelStarted = false
           runnerCount =0
            applesCollected = applesCollected + 1000
            waveCount = 1
            initializeLevel()
         elseif(waveCount==10 and waveFinished == true) then
            print("You Survived!  Game Over")
            --  Add a splashy win screen
         elseif(waveFinished) then
            applesCollected = applesCollected+ 100
            waveCount =waveCount +1
            levelStarted= false
             runnerCount=0
            startWave(waveCount)
         end
      end
end
```

Noises Off!

This routine is pretty straightforward. If the player presses the sound icon, it will mute or play sounds. Sound is initially set to true. So the first time it is called, it will be changed to false. Anytime a sound is played, we first check the sound variable to see if it is true or false.

```
local soundOnOff = function()
      if sound then
            sound=false
      else
            sound=true
      end
end
```

Suspense is Killing Me! - adding suspend/resume/save options

One of the requirements in submitting a game to the iTunes store is that it must be pauseable so that, should the player receive a phone call or hit the home button, the game can be suspended.

Pausing the physics engine is easy. You tell it to pause. Done. However, transitions are a little trickier. Espeically if you want to resume. There are several sample transition routines available on the Corona website, but none of them worked correctly with my sprites. Often the transitions would fail to resume. Thus I found it necessary to write my own routine to handle transition pausing.

You might remember earlier in the chapter when we created a variable for the runner called baseTime. It recorded how long it should take to get to the next waypoint. BaseTime is decreased every loop of the gameLoop function. Thus when it is time to resume the transition (which requires making a new transition for the runner) we have a fairly accurate guess as to how long it should take to get to the next waypoint (within 400 milliseconds).

The first section of the routine handles coming out of a paused state (i.e. resume). It restarts the physics engine so that collisions can occur. Then the resume goes through each runner. Any runner that is not knocked down or already removed receives a new transition.to command. This is very similar to the routine in pathfinder. The only difference being that time is based off of the stored baseTime.

```
local function pauseGame()
   if paused then
      physics.start()
      paused = false
      for i=1,#runner do
         if(runner[i].myName ~= nil or runner[i].myName ~="down") then
            runner[i].currentTransition = transition.to(runner[i], {x=
waypoint[runner[i].nextWP].x, y = waypoint[runner[i].nextWP].y, time =
runner[i].baseTime})
            runner[i]:play()
         end
      end
   else
```

If we need to pause, then we start with the physics engine. After physics are paused, then we loop through all the runners and cancel the transition. We also pause the sprite animation (it looks a little strange to have them running in place).

```
      physics.pause()
      paused = true
      for i=1,#runner do
         if(runner[i].myName ~= nil or runner[i].myName ~="down") then
            transition.cancel(runner[i].currentTransition)
```

```
            runner[i]:pause()
         end
      end
   end
   return true
end
```

The start game function does just that; it starts the game! It begins by removing the splash screen and the buttons. Then a call is made to initialize the level and status displays. The last feature of this function is to start the timers and event listeners.

```
local startGame = function()
      splash:removeSelf()
      newGameButton=nil
      loadGameButton=nil
      initializeLevel()
      initializeStatus()
      timer.performWithDelay(tick, gameLoop, 0)
      sndImage:addEventListener("tap", soundOnOff)
      pauseImage:addEventListener("tap", pauseGame)
      Runtime:addEventListener("collision", pathfinder)
end
```

If the player closes the game, a copy is saved to the documents folder on the device. I didn't attempt to save every piece of data (you could if you wanted to). Load can only occur as an option if the savedGame.txt file exists, so I did not perform the normal check here. The check is handled by the splash screen. Instead, we can proceed to open the file as read-only and read all the data into saveData.

```
local loadGame = function()
      -- Set location for saved data
      local filePath = system.pathForFile( "savedgame.txt",
system.DocumentsDirectory )
      local file = io.open(filePath, "r")

      local saveData = ""
      saveData = file:read("*a")
      --print(saveData)
      file:close()
```

Using the string.gmatch command, I saved all of the data to an array called savedData (I know.. real original name). Once the data is parsed, we can load the variables we are interested in from the saved data. As numbers are converted to strings (unless you specifically give the command to save them as numbers) automatically, it is necessary to convert the string back to a number with tonumber.

```
      --break the data into usable info
      savedData = {}
--    print( string.find(saveData, ","))
      for k, v in string.gmatch(saveData, "(%w+)=(%w+)") do
```

```
            savedData[k] = v
--          print(k..", "..savedData[k])
      end
      level = tonumber(savedData["level"])
      waveCount = tonumber(savedData["waveCount"])
      score = tonumber(savedData["score"])
      applesCollected = tonumber(savedData["applesCollected"])
      damage = tonumber(savedData["damage"])
--    print (level..", "..waveCount)
      startGame()
end
```

I suppose you're not surprised to find a save game function considering we just looked at a load game function. If you are: surprise!

This function simply opens the file for writing, destroying any previous data written. I choose to only save the current level, wave, score, applesCollected and damage. It would be possible to also save the current tower layout if you were so inclined.

```
local saveGame = function()
      -- Set location for saved data
      local filePath = system.pathForFile( "savedgame.txt",
system.DocumentsDirectory )
      local file = io.open( filePath, "w+" )
      file:write("level="..level..", ")
      file:write("waveCount=".. waveCount..", ")
      file:write("score=".. score..", ")
      file:write("applesCollected=".. applesCollected..", ")
      file:write("damage=".. damage..", ")
      file:close()
end
```

OnSystemEvent is here to handle any, well, system events. Should the app suddenly close or exit, this function will call pause and save. If the user resumes the app after doing something else, it will call pause again, which will resume the suspended game.

```
local function onSystemEvent(event)
      -- handle unexpected close or interruptions
      if (event.type=="applicationExit" or
event.type=="applicationSuspend") then
            --if game isn't paused, pause it
            if (not paused) then
                  pauseGame()
            end
            -- save game
            saveGame()
      elseif(event.type=="applicationResume") then
            pauseGame()
      end
```

```
end
```

It's a Splash - add splash screen

Almost finished! The startMenu function is the first function called in the game. It displays our splash screen (centered to the device screen) and two buttons: the New Game button, and a Load Game button if there is a previously saved game.

```
local startMenu = function()
      -- Show splash screen
      splash = display.newImage("splash.png")
      splash.x = display.contentWidth/2
      splash.y = display.contentHeight/2
      newGameButton = ui.newButton{
            default = "buttonBlue.png",
            onPress = startGame,
            text = "New Game",
            emboss = true
      }
      newGameButton.x = display.contentWidth/2
      newGameButton.y = display.contentHeight/2 - 50

      -- Check for old game (if one exists)
      local filePath = system.pathForFile( "savedgame.txt",
system.DocumentsDirectory )
      local file = io.open(filePath, "r")
      if file then
            -- Game exists, so show load button
            loadGameButton = ui.newButton{
                  default = "buttonBlue.png",
                  onPress = loadGame,
                  text = "Load Game",
                  emboss = true
            }
            loadGameButton.x = display.contentWidth/2
            loadGameButton.y = display.contentHeight/2 + 50
      end
end
```

And finally, we call the startMenu function to get everything going. This is also a good place to add the OnSystemEvent listener.
I have commented out a very useful routine from spriteloq which allows you to profile the app performance on your system. Very useful for tweaking game performance!

```
startMenu()

Runtime:addEventListener( "system", onSystemEvent )
--require('loq_profiler').createProfiler()
```

Summary

As we wrap up this final instructional chapter, I want to take a moment to again thank Graham Ransom for providing a lite verison of Lime and Brandon Burton for creating the sprites. Also, the great people at SpriteLoq for including a 30-day free trial with Corona and the makers of Tiled. Thank you! Please support each of these individuals or companies by purchasing their products.

I have been asked by many people to create a book that discussed the full game development pipeline for a game like Rotten Apples. To not disappoint, I am currently writing a book with my graphics person that will cover the entire art and game development pipeline for this project. Look for it on my website (http://www.BurtonsMediaGroup.com/books), Amazon, iBookstore, or Kobo by mid-2012.

As you continue to explore Corona, please recognize the wonderful developer's community that is available online at http://www.anscamobile.com.
The Corona community is also very active on twitter. Check out #CoronaSDK, @ansca, @carlosicaza, @walterluh, or my tweets: @DrBrianBurton to learn the latest features coming to Corona.

Finally, I only had space to share a few great resources and tools that are available for Corona. There are many more, and I have included a partial list in Chapter 17.

Assignments

1) Add additional runner types to the game.

2) Add a win and lose game over screen.

3) Adjust the tick timer, apples collected with each knockdown and other variables. How does this impact game play?

4) Add random power-ups that when tapped give the player a special feature.

5) Create your own level using Tiled. Use the included sample levels to help set the properties.

Chapter 17
Additional Resources

Over the past couple of years there have been some really great supporting applications and tools to make developing with Corona even faster and easier. I am including a sampling of these products with my thoughts. I am not associated with any of these companies. Some of these resources are free; others are available for a small amount of money. I am sure that I have left-out software that is very worthy of being included in this list. If you find a great tool that should be included, please let me know!

- Autocomplete
- BBEdit
- Corona Comic
- Corona Project Manager
- Corona Remote
- CoronaUI
- Crawlspace
- Director
- Kwik
- LevelHelper
- Lime
- Physics editor
- SpriteHelper
- Spriteloq
- Texture Packer
- Tiled
- Useful Websites

Autocomplete

Autocomplete is a great resource that can be a real time saver. While it isn't truly called autocomplete, it is a library of autocomplete commands that can be added to various editors such as Corona Project Manager. While it is the work of many contributors, the autocomplete resource list is currently being maintained by Lars Nordstrom. It is simple to use. Just copy the appropriate list into the autocomplete list in your editor (in Corona Project Manager, select Preferences > Editor, then cut and paste into Autocomplete Words).
Vendor: Creativefusion (with contributions by many others)
Cost: Free
Website: https://github.com/lano78/AutoComplete-CPM

BBEdit

While only available for the Macintosh, this is one very useful editor. I am now to the point where I choose to do all of my Corona script writing and editing on my Macintosh laptop (even though the keyboard isn't as nice) instead of my PC because of BBEdit. As their website says: "It doesn't suck."
Vendor: Bare Bones Software Inc.
Cost: $99.99, 30 day trial available
Website: http://barebones.com

Corona Comic

If you have ever wanted to make your own comic book, you can now easily create one! The Corona Comic SDK is available in the Code Exchange section of the Ansca Mobile website. The SDK comes with a nice sample app and is easy to use. Just take your image files from Photoshop (or any photo editor), follow the directions in the SDK, and you have your own comic ready for distribution!
Vendor:
Cost: $ free
Website: http://developer.anscamobile.com/comics

Corona Project Manager

Corona Project Manager is a must have if you are going to be a serious Corona developer. It keeps your projects organized, allows you to import assets from other projects, keep a code library, view assets, open them in outside editors, and launch the simulator. It has built-in features to help you configure your build.settings and config.lua files with just a few clicks.
Vendor: J.A. Whye
Cost: $75
Coupon: Use the code DRBOOK on the second page of the order form to receive a 30% discount. A 30 day trial is included with your subscription.
Website: http://coronaprojectmanager.com

Corona Remote

Corona Remote is a sweet little app that sends data from your iPhone back to the Corona simulator, saving you time in not having to re-publish your app during your app testing. Corona Remote returns both accelerometer data and compass data to your testing system.
Vendor: Matthew Pringle
Cost: $9.99
Website: http://www.coronaremote.com

Crawlspace

Crawl space is the swiss army knife of app development. It is a free download made available by Adam Buchweitz of Crawl Space Games. The features list reads like a wish list for most developers: Timers, popups, background music control, table printing, paragraph control for text, help functionality, fade in and out; the list of features just goes on and on!
Vendor: Adam Buchweitz
Cost: $ donation
Website: http://www.crawlspacegames.com/crawl-space-corona-sdk-library/

Director

Sometimes simple is best. Ricardo Rauber has accomplished creating an awesome class of routines to manage scenes and transitions in your apps. It is one of the most popular downloads in the Corona Community Code Exchange area. He has made it freely available. If your app will have multiple views or screens, this is a must have!
Vendor: Ricardo Rauber
Cost: $ Donation
Website: http://developer.anscamobile.com/code/director-class-10

Kwik

Kwik is an Adobe Photoshop plugin that allows you to add animation, videos, sounds, rollover effects, buttons, and much more, all in Photoshop. You can then export the results as a finished mobile app. This product is what the artists have been waiting for! Now they can create the complete app in Photoshop with no coding (though you can still edit the resulting code). If this catches on, what will the coders have left to do?
Vendor:Kwiksher
Cost: $49.99
Website: http://www.kwiksher.com

LevelHelper

"Wow!" was my first thought when I opened up LevelHelper. After playing with level helper for just a few minutes, I began to get seriously excited about the next sprite based game that I would be making (LevelHelper will be a featured app in my next book on Game Development with Corona). Combined with SpriteHelper, you can take some serious time off your development cycle! At only $16.99, you would be silly to not use such a great resource and time saver.
Available through the Mac App store.
Vendor: Bogdan Cladu
Cost: $16.99
Website: http://www.levelhelper.org

Lime

One of the earliest tools that I started using with Corona was Lime by Graham Ransom. Graham has created a great resource in Lime that handles the management of tile-based games. I showed Lime and Tiled (the FREE tool for making the basic tiled environment) to my student about 2 weeks before their final projects were due. Several of my students, upon seeing what could be done with Tiled and Lime, immediately changed their projects to one that was built upon Lime and Corona.
Overall I have been very impressed with the Lime package. It is still in beta, and has a great beta price of £20.
Vendor: Monkey Dead Studios, LTD; Graham Ransom
Cost: £20 (beta)
Website: http://www. justaddli.me

Physics Editor

Have you ever spent far too much time trimming and cropping an object? I know I have, and I really don't have time to waste on such a basic function. Then there is the whole concave/convex concern for physics engines! Physics Editor removes all those concerns. In just a drag and few clicks you have a trimmed object ready for import into your project. Nice and easy!
Physics Editor is available for Mac and PC. Texture Packer and Physics Editor are available as a package.
Vendor: code'n'web – Andreas Löw
Cost: $19.95
Website: http://www. www.physicseditor.de

SpriteHelper

SpriteHelper is the Robin to LevelHelper's Batman. With SpriteHelper you can quickly configure the physics and textures for your sprites to plugin to LevelHelper. These two programs go hand in hand, and at these great prices, two hands are better than one!
Vendor: Bogdan Vladu
Cost: $11.99
Website: http://www.spritehelper.org

Spriteloq

So I was writing the end of chapter 5 and found that I had no unique sprite sheets to use for the demonstration. As I've mentioned before, I'm no artist. I had some simple animations in Flash, but nothing setup as a sprite sheet. In steps Spriteloq to the rescue! In a very short time I had exported my animation from Flash and converted it to a sprite sheet. Awesome tool and a great time saver!
Vendor: Loqheart
Cost: $49.00 (introductory price); free 30 day trial

Website: http://www.loqheart.com/spriteloq

Texture Packer

Texture Packer takes your sprites and packs them, with incredible results! It is capable of dramatically reducing your file size and packing more sprites into a single sheet. Texture Packer also will perform trimming, removing excess transparent pixels from the borders of your sprites. If you're working with sprites for your games, don't pass up this great tool! Available for Mac and PC. Texture Packer and Physics Editor are available as a package.
Vendor: code'n'web – Andreas Löw
Cost: $basic – free; Pro - $19.95
Website: http://www.texturepacker.com

Tiled

Tiled is a free tool for making sprite based maps. When combined with Lime, you will have your sprite based game up and going in no time! Tiled is available for Windows or Mac.
Vendor: Thorbjørn Lindeijer
Cost: $Donation
Website: http://www.mapeditor.org

Useful Websites

Throughout this book I have not listed and explained every API and possible parameter for the API's that are available. For a complete list of APIs available for Corona see: http://developer.anscamobile.com/resources/apis/

Free Isometric images

For free isometric images visit Reiners tilesets: http://www.reinerstilesets.de

Music

JewelBeat provides 99¢ music and music loops that can be used in your apps and games. http://www.jewelbeat.com/

Sound effects

http://www.flashkit.com/soundfx/

Tutorials

http://www.learningcorona.com – This is a great site with links to tons of tutorials, sample code, as well as links to 3rd party tools and libraries.

Appendix A
The Lua Language

This appendix on the Lua scripting language was supplied by the great people at Ansca Mobile.

Lua
An Introduction

The Corona platform uses Lua as its interface between the developer and its framework. In order to use Corona, you'll need to become fairly comfortable with Lua. Thankfully, however, this is a good thing, as I'm sure you'll discover in the following appendices.

What is Lua?

Lua is a scripting language created in 1993 by the Computer Graphics Technology Group (Tecgraf) of the Pontifical Catholic University of Rio de Janeiro, Brazil. Its chief architect is Roberto Ierusalimschy, who you may recognize from the introduction to this book. According to Ierusalimschy, Lua semantics were heavily inspired by Scheme, although its syntax is quite different.

Lua is a free, open source language, covered by the MIT license, which means it has the loosest and least restrictive usage requirements in the software development industry. Due to this level of freedom and its maturity, gained from its years of dedicated followers, Lua has grown to be one of the smallest, fastest, and most flexible languages of its kind. Its extreme level of portability makes it prevalent on almost any hardware device, while its speed and stability have seen it used in the most demanding commercial applications. For example, significant portions of Adobe Photoshop Lightroom are written in Lua.

Lua has also become known as the scripting language of choice for game developers. One of its earliest commercial applications was the LucasArts adventure game Grim Fandango (1998), and it has gained wide public exposure as the scripting interface in Blizzard's World of Warcraft. Lua has also been used for scripting in Crytek's CryENGINE2 (used in Far Cry) and Garry's Mod for Half-Life 2, and for internal logic in games like Psychonauts, Heroes of Might and Magic VI, S.T.A.L.K.E.R: Shadow of Chernobyl, Bioware's MDK2, and EA's SimCity 4.

Because Lua is small, fast, and extremely portable, with a full Lua bytecode interpreter generally needing just 150 KB, it has also become a natural for embedded applications. Lua scripting interfaces are available for the Wireshark network analysis program and the Logitech Squeezebox Duet audio player; Lua is used in firmware development for Olivetti

printers and included with the "Canon Hack Development Kit" for Canon PowerShot cameras; and the eLua Project maintains a full version of the language designed to run directly in microcontrollers.

Recent mobile platforms like iOS and Android resemble a cross between embedded platforms and game consoles. Unsurprisingly, many of the top mobile game companies like Electronic Arts and Tapulous use Lua in their products, and it can currently be found in bestselling iPhone games like Angry Birds, Tap Tap Revenge 3 and Diner Dash.

However, while Lua executes very quickly for a scripting language, it is necessarily slower than a compiled native language. For this reason, console or mobile games employing Lua will typically use it for tasks like level design, interface layout, game logic, scripted events, enemy AI, or adaptive audio, with native languages like C, C++ and (on the iPhone or iPad) Objective-C reserved for more computationally intensive elements like graphics rendering and physics engine calculations. Lua is designed to work within a "host program" written in C/C++, and this two-tiered architecture allows for rapid prototyping, development and revision of games, without sacrificing native speed and responsiveness in the final result.

The goal of Corona is to bring this professional game engine architecture to a wider group of developers, in an easy to use, cross-platform mobile development tool. Ansca Mobile have taken a lot of time to create a Lua layer that maps cleanly to underlying native frameworks written in C, C++ or Objective-C for iOS and Android. These include support for hardware-accelerated OpenGL graphics; touchscreen gestures; device sensors like the accelerometer, gyroscope, compass and GPS systems; and specialized game-development tools such as the Box2D physics engine.

What's more, the Lua methods created to expose this functionality follow a much simpler API, dramatically reducing the amount of coding required to get things done, even though most of the final compiled application will consist of native code.

Lua in Practice

Lua is a relatively simple language with some very powerful features. As a games developer, you likely won't need many of the more complex features of Lua, and can probably get by with the basics of the language. In this book, we'll be covering everything you need to know to get the job done, but if you'd like to learn about Lua in more depth, the official Lua book, *Programming in Lua*, is available to read online at http://www.lua.org/pil/ and is a very good resource.

Types and Variables

Like all languages, Lua makes use of datatypes, which are structures representing your application data. Unlike many other mature languages, Lua has very few standard types.

This is mainly due to Lua's sheer level of flexibility. In Lua, most of the tasks you need to carry out can be performed with objects that build on these standard types. They include :-

- Nil
- Numeric values
- Strings
- Boolean values
- Tables
- Functions
- Userdata
- Threads

In this book, we will not be discussing Userdata and Thread types as, apart from falling into complexity beyond the scope of this book, you really won't be needing them (or be able to use them) with Corona or iPhone development in general. That leaves us with only six standard types. Certainly not many!

Seasoned developers may be wondering why functions are listed here as a datatype. The truth is that, in Lua, a function is considered a first class citizen in the sense that it can be stored in a variable. This permits the use of Lua functions as closures, which means that functions can return other functions, or pass them to other objects; this is very handy for callback methods. We'll be discussing functions later in this appendix, and closures in the following appendix.

Type Declarations

Lua is a dynamic language. This means that you do not get compiler type checking when writing your applications; however, it also means that you get a certain level of flexibility when performing operations on your data, as it is often possible to substitute data types. For example, you could concatenate a number to the end of a string, or use a string representation of a number in a math function.

When using a variable for the first time, you simply assign it a value. The Lua interpreter knows when a variable is first created, so it handles all the necessary overheads for you:

myVar = "some value"

The above is an example of how you would create a new variable. Notice how we have not needed to specify a type. Once a variable is created, you'll need to remember the type of data it contains yourself, though Lua will not normally complain if you choose to do something funky with it.

Also, note that we haven't had to end our declaration with a semi-colon or some such character. For the most part, Lua uses the end of the line of code to end an expression, though there are a few occasions when this is not true.

The following is an explanation of Lua's standard types. Due to their complexity, however, the Function type will be discussed later in this appendix and Tables in the next, when you should have a better grasp of the language.

Nil

The Nil type represents a nil value or null in some languages. In essence, it is a lack of an object. This is different to a lack of value as, in some cases, an empty object might exist, which would still be considered not nil. Nil is useful for passing between or from functions as a way to state that a value does not exist as it is the definitive means for type checking such situations.

Booleans

Booleans are values which are either true or false. You will often use Booleans as the return value of a function or comparison between two values. For example, you might check that a value is not Nil.

```
result = ( myValue ~= Nil )
```

If the value is Nil, then the result variable would contain False, otherwise it would contain True.

In boolean expressions, such as 'if' statement signatures, all objects and values are considered true, with the exception of False and Nil, which are considered false.

Numeric Values

Numeric values include all possible literal numeric values. In Lua, all numeric values are floats. When dealing with integers such as 1, 5 or 1005, you are really dealing with 1.0, 5.0 and 1005.0 respectively.

Numeric values in Lua can be represented using several different types of notation. Beyond the standard base 10 notation, Lua also supports scientific notation and hexadecimal. Scientific notation is where a number is represented with the letter 'e', either upper or lower case, embedded within it. The 'e' allows the value to be shortened by specifying a number of added zero's to the right hand side of the number if the value to the right of the 'e' is positive or by moving the decimal point so many values to the left if the number to the right of the 'e' is negative. The following are examples using scientific notation:

```
print (9e3)
-- outputs 9000
print (22e-4)
-- outputs 0.0022
```

An alternative notation is hexadecimal. Hexadecimal is base 16 numeric values. This means that, rather than counting in steps of 10, numbers count in steps of 16, where the numbers 11 through 15 are represented by the letters 'a' through 'f'. In order to alert the interpreter

that we're dealing with base 16 values, hexadecimal values are prepended with the characters 0x. Here are some examples:

```
print ( 0xf )
-- outputs 15
print ( 0x55 )
-- outputs 85
```

Numeric Operators

Like many other languages, Lua provides many of the standard operators for working with mathematical equations:

```
-- addition operator
print (2+2)
-- subtraction operator
print (10-5)
-- division operator
print (8/2)
-- multiplication operator
print (5*3)
-- exponent operator (the power of)
print (5^2)
```

Lua calculates values from left to right. When working with mathematical equations, you can enforce values to be calculated first by wrapping them in parenthesis. When Lua finds a mathematical equation, it always calculates the inner most equation contained within parenthesis first, then works its way to the outer nested equations. For example:

```
print (2 + 2 * 2 + 2)
-- will be different from
print ((2 + 2) * (2 + 2))

print (3 + 3 * 3 * 3 + 3)
-- will be different from
print ((3 + 3 * 3) * (3 + 3))
-- which will be different from
print (((3 + 3) * 3) * (3 + 3))
```

Dividing by Zero

In Lua, all mathematical expressions return a numeric value, with the exception of equations where a number is divider by zero. For instance, take the following expression:

```
print (5/0)
```

In many languages, this expression would raise an error. However, in Lua as used by Corona, this expression would print the value 'Inf', meaning infinite. 'Inf' is not usable in a numeric equation and thus should be caught where possible in a value check. This should

preferably be carried out before the equation by checking if the divisible value is zero. However, you can also check the result of the equation by doing the following:

```
result = 5/0
print ( result == 1/0 )
-- outputs true
```

Lua provides quite an extensive number of math functions. We'll be looking at these a little later in this section.

Strings

Strings are sequences of characters, such as letters, numbers, punctuation symbols and even control characters (tabs, newlines etc). They include any data that exists between a pair of quotes.

Quoting Strings

There are three types of quote you can use to contain strings. These are single quotes:

```
'This is a string'
```

double quotes:

```
"This is also a string"
```

or even double square brackets.

```
[[This is a square bracketed string]]
```

Square brackets represent literal strings. While in single or double quoted strings, the compiler will look for special characters, such as escape sequences or control characters, everything contained within square brackets are treated exactly as they're given. Thus, if a tab is used within the string, it will be treated by the compiler as a literal tab and will be visible when output to the user.

Square brackets are useful for working with strings that span more than one line as in, if you try to span single or double quoted strings on more than one line, you will incur an error:

```
[[This is
a string that
spans multiple
lines]]

"This string
will raise an
error"
```

It is possible, however, to enforce multiple line traversal with single or double quoted strings by escaping the end of each line:

> *"This string *
> *will no longer *
> *raise an error"*

Escaping Characters

When choosing the type of quotes for your string, note that you can include quotes of a different type within that string, while matching quotes will need to be escaped with the backslash character.

> *"This is 'fine for quoting' with single quotes"*

> *"This \"needs escaping\" to be legal"*

> *'This isn\'t always obvious'*

Control characters also use escaping to alert the compiler of their difference to alpha numeric characters. Control characters can include tabs (\t):

> *"\tThis string starts off indented"*

vertical tabs (\v):

> *"\vThis string will have padding above it"*

and newline characters (\n):

> *"This string\nwill appear on two lines"*

Another type of character that requires escaping is the backslash itself:

> *"Only one of these \\ will be visible"*

Concatenating Strings

Concatenation is the means to join two separate strings into a single string. Concatenation is done using the concatenation character '..'; two periods, side by side, without any spacing between them. When concatenating strings, the original strings remain unchanged. For example:

```
strOne = "abc"
strTwo = "def"
strThree = strOne .. strTwo
print (strOne)
-- outputs abc
print (strTwo)
-- outputs def
```

```
print (strThree)
-- outputs abcdef
```

If you would like a space to appear between the two strings, then be sure to add it to one of the strings, or add it separately when concatenating:

```
strThree = strOne .. ' ' .. strTwo
print (strThree)
-- outputs abc def
```

Comparing Values

Values can be compared for sameness using Lua's comparison operators. These include:

==	is equal to
~=	is not equal to
>=	is greater or equal
<=	is less or equal
>	is greater than
<	is less than

Each of the comparison operators accepts two parameters; one to the left of the operator and one to the right. The operator performs the comparison and returns the result as a Boolean value. For instance:

```
result = ( val1 < val2 )
```

In the above example, if the value of val1 is smaller than the value of val2, then the result variable would contain True. Otherwise, the result variable would contain False.

Sometimes, it will be necessary to compare the type of values rather than the actual values themselves. This is a very common requirement in dynamic languages such as Lua when you're never really sure what type a variable holds. In order to achieve this, the type of the value needs to be extracted into a readable format. We'd then use the extracted data in our comparison expression.

Acquiring a values type in Lua is performed using the 'type' function. 'type' accepts a single parameter - the value of the type to extract - and returns a representation of the type as a string. Thus, when used in a comparison expression, we could do the following:

```
result = type( myVar1 ) == type( myVar2 )
```

If the type of the variables myVar1 and myVar2 are both strings, then the result will contain True.

We'll look more at comparison expressions later, when we discuss conditional statements.

Boolean Operators

While comparison operators return a Boolean value from data pairs, Boolean operators perform the task of returning a Boolean value from Boolean pairs.

In the loosest sense, you'll rarely want to use comparison operators with Boolean values as those values are already Boolean in nature and will be the same type as the return value. In this instance, performing a comparison serves little purpose. However, what you will want to do will be to perform logic based on the outcome of more than one comparison result by combining those results into a single Boolean value. Boolean operators allow you to do just that.

The and Operator

The 'and' operator accepts two Boolean values (or Boolean returning expressions) and returns true if, and only if, both values are true. Thus, the following will ensue:

```
print ( true and true )
-- outputs true
print ( false and true )
-- outputs false
print ( true and false )
-- outputs false
print ( false and false )
-- outputs false
```

The or Operator

The 'or' operator accepts two Boolean values (or Boolean returning expressions) and returns true if both or either value is true. So, using the same approach, we would get:

```
print ( true or true )
-- outputs true
print ( true or false )
-- outputs true
print ( false or true )
-- outputs true
print ( false or false )
-- outputs false
```

The not Operator

The 'not' operator is the final Boolean operator and is used to negate a Boolean value. This means that, if used with the value True it will return False, while if used with False, will return True. The 'not' operator can only be used with a single value to negate an expression. The expression itself will need to be contained within parenthesis.

```
print ( not false and true )
```

```
-- outputs true
print ( not ( false or false ) )
-- outputs true
```

Stacking Boolean Operators

Although the ‘and’ and ‘or’ Boolean operators only work with two Boolean values at a time, it is possible to stack these expressions. This is because the Lua virtual machine calculates a given Boolean expression, creating a Boolean result, which it then applies to the next Boolean expression. This occurs in a left to right direction. Therefore, in the following example:

```
myVar1 = 1
myVar2 = 2
myVar3 = 3
result = myVar1 < myVar2 and myVar3 > myVar1 and ( myVar1 + myVar2 ) == myVar3 or
myVar1 > myVar3
```

The first expression, “myVar1 < myVar2”, evaluates to true. The result of this expression is then compared to “myVar3 > myVar1”, which also results in True. Using the ‘and’ operator with these results also gives the value true. This value is then compared to “(myVar1 + myVar2) == myVar3) which is true and, using the ‘and’ operator, also returns true. Finally, that value is compared to the expression “myVar1 > myVar3”, which is false, using the ‘or’ operator. As “true or false” results in true, the result variable now also contains true.

If the above example seems a little messy and hard to follow (I know it is for me), you can make the full expression more clear by wrapping each nested expression with parenthesis, like this:

```
result = ( (myVar1 < myVar2) and (myVar3 > myVar1) and (myVar1 + myVar2 == myVar3) )
or (myVar1 > myVar3)
```

Lua Data Functions

Lua provides a number of objects and functions for working with data. Many of them will be equivalent to functions of the same name in other languages.

String Functions

Lua provides a large number of functions for working with strings. Most of these functions belong to the ‘string’ object. We’ll be working with objects later, but for now, just know that the functions you will be looking at will start with ‘string.’ (the word string, followed by a period).

Finding the Length of a String

Very often, you’ll want to know how long a string is. This is certainly useful in Corona as you’ll know how many characters in a given font will fit on the screen of an iPhone. As a result, knowing how long a string is will help with laying out your text on the screen.

In Lua, you have a couple of options for finding the length of a string. The first of these options is by using the ‘len’ function:

```
str = "The quick brown fox jumped over the lazy dog"
print ( string.len(str) )
-- outputs 44
```

The alternative to string.len is to use the length operator ‘#’. The length operator can be used with other objects, but is particularly useful with strings and tables. When used with strings, you simply place the operator before the string you wish to query for its length, and Lua will do the rest:

```
print ( #str )
-- outputs 44
```

Global Substitution

Global substitution is the process of replacing all occurrences of a given pattern. In Lua, we perform global substitution using the ‘gsub’ function. For example, we might like to replace the letter ‘o’ in a sentence with another character:

```
print ( string.gsub( str, "o", "@" ) )
-- outputs The quick br@wn f@x jumped @ver the lazy d@g    4
```

Note the number 4 at the end of the output. This is not a typo. Lua actually returns the number of replaced instances of the pattern within the string as well as the resulting string itself. Being able to return more than one value from a function is a powerful feature of Lua that we’ll examine more of later in this appendix.

Global substitution can also be used to replace whole words or sentences. You can also specify a special pattern string that works similarly to regular expressions in other languages, though not quite so powerful. For example, if we wanted to replace any five letter words beginning with the letter b with the word “red”, we could use the following code:

```
print ( string.gsub( str, “b....”, “red” ) )
-- outputs The quick red fox jumped over the lazy dog
```

The period symbol, when used in a pattern, represents a wildcard character, so will match any character in the string. In the previous example, we simply said “replace any five characters that begin with the letter ‘b’ with the word ‘red’. The wildcard character will also match spaces in your strings.

Patterns in Lua is quite an extensive subject and beyond the scope of this appendix. For a more thorough explanation of patterns, refer to the official Lua manual.

Finding a Pattern in a String

Occasionally, you may like to know where a particular string of characters exists within a given larger string. For instance, you might like to locate potential phone numbers or offensive words. You can perform this process by using the 'find' function:

```
print ( string.find(str, "brown") )
-- outputs 11    15
```

The result of the function is the location of the starting character for the match as well as the location of the ending character.

The 'find' function returns the first match only, so further matches will need to be requested using a starting location beyond a previous match. For example, our previous match ended at character 15. Therefore, we could repeat the search for the word 'brown' from character 16 onwards. We do this by providing a third parameter; the starting location.

```
print ( string.find(str, “brown”, 16) )
-- outputs nil
```

This time, no result was found as the search word did not reoccur. Thus, the function returned 'nil'.

As with global substitution, you can also use a pattern to find a broader range of possible character matches.

Matching a Pattern in a String

As well as finding the location of a string of characters in a string, Lua also lets you find which words in a string match a given pattern. You perform this task using the 'match' function. The parameters for 'match' are the same as with 'find'. However, rather than returning the location, 'match' returns the actual word found. For example:

```
print ( string.match(str, "b....") )
-- outputs brown
```

As with 'find', match also accepts the starting location as its third parameter.

When using 'match', you will use a pattern as the search criteria, quite simply because you will already know which word will match when using non-pattern based searches.

Obtaining a Characters Byte Value

You can acquire the ASCII value (American Standard Code for Information Interchange) of any character in your string by using the 'byte' function. For example, to get the ASCII value of the fifth character, you would use:

```
print ( string.byte(str, 5) )
```

```
-- outputs 113
```

ASCII characters are useful when you want to check for a characters actual value in memory as opposed to its visual representative value. For instance, you may want to check that a character is an ‘o’ rather than a ‘0’ (*oh* rather than zero), or you may want to compare accented and non-accented letters.

Getting a String Value from Bytes

As well as getting ASCII values from characters, Lua also provides the means to get string characters from ASCII values, using the ‘char’ function:

```
print ( string.char(113) )
-- outputs q
```

The ‘char’ function can take multiple parameters, so it’s possible to return a whole string from a number of ASCII codes:

```
print ( string.char(65, 66, 67) )
-- outputs “ABC”
```

Changing the Case of Characters

In the previous examples, you saw how to search for and extract characters in a string, but what would happen if you were to run the following example?

```
str = “The quick brown fox jumped over the lazy dog”
print ( string.find( str, ‘t’ ) )
```

One would assume the result printed to screen would be the number 1, matching the ‘t’ from the word ‘The’. If you guessed this, however, you’d be wrong. Instead, the output will be 33, matching the ‘t’ from the second occurrence of the word ‘the’. This is because the lowercase letter ‘t’ and the uppercase letter ‘T’ are considered separate characters and are therefore unequal.

When performing such searches and you are not bothered about the case of the letter or word you wish to find, it helps to convert the string to consist of all lower or upper case letters first. This is performed using the ‘lower’ and ‘upper’ functions, respectively. For example:

```
print ( string.lower(str) )
-- outputs the quick brown fox jumped over the lazy dog
print ( string.upper(str) )
-- outputs THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG
```

Retrieving a Segment of a String

Occasionally, one might want to extract a segment of a string as a new string. For example, we may want to acquire the chunk of characters starting from the beginning of the string

and ending with the word 'brown'. Now, we could ascertain that the word 'brown' ends at character 15 using the 'find' function, but to extract the text before and including that location, we'd need to use the 'sub' function:

```
print ( string.sub(str, 0, 15) )
-- outputs The quick brown
```

The word 'sub' is short for sub-string.

Math Functions

Lua includes numerous functions for performing both simple and complex mathematical calculations. Some of these you may want to use in your Corona applications for dealing with animation or working out a players score, etc.

Lua's math functions belong to the 'math' object. We'll be looking at objects later in this appendix. Be aware that when invoking a math function it should be preceded with 'math.' (the word math followed by a period symbol).

The following table lists a number of the functions provided by Lua's 'math' object.

Function	Parameters	Returns
abs	number	The absolute value of [number]
acos	cosine (-1 to 1)	Returns an angle in radians (0 to pi) of the [cosine]
asin	sine (-1 to 1)	Returns an angle in radians (-pi/2 to pi/2) of the [sine]
atan	tangent	Returns an angle in radians (-pi/2 to pi/2) of the [tangent]
atan2	number (x), number (y)	Returns the angle theta (-pi to pi) of the point (x, y)
ceil	number	Returns the nearest integer greater than or equal to [number]
cos	[angle] in radians	Returns the cosine of [angle]
cosh	[angle] in radians	Returns the hyperbolic cosine of [angle]
deg	angle in radians	Returns the number of degrees in [angle]
exp	[exponent]	Raises base-e to the [exponent]
floor	number	Returns the nearest integer smaller than or equal to [number]

Function	Parameters	Returns
fmod	numerator, denominator	Returns the remainder of [numerator] / [denominator]
frexp	number	Returns the mantissa and the exponent as an integer of [number]
log	number (greater than 0)	Returns the base-e of [number]
log10	number	Returns the base-10 of [number]
max	number, number	Returns the larger of the two given numbers
min	number, number	Returns the smaller of the two given numbers
modf	number	Returns the integer and decimal part of [number]
pow	power, exponent	Raises [power] the [exponent]
rad	angle in degrees	Returns the number of radians in [angle]
random	lower limit, upper limit	Returns a pseudo-random number between [lower] and [upper]
randomseed	number	Seeds the random number for the random function
sin	angle in radians	Returns the sine of [angle]
sinh	angle in radians	Returns the hyperbolic sine of [angle]
sqrt	number	Returns the square root of [number]
tan	angle in radians	Returns the tangent of [angle]
tanh	angle in radians	Returns the hyperbolic tangent of [angle]

A Note About Code Blocks in Lua

While Lua syntax is quite similar to languages like JavaScript or ActionScript, one difference you'll notice right away is that it uses keywords like "then" and "do" rather than braces, to define code blocks. This is because Lua reserves the brace characters for declaring tables. Tables are discussed in detail in the next appendix.

Conditional Statements

Okay, so now you can work with Lua's variables, but programming is all about making decisions. Based on the content of variables in your application, you may want to trigger a particular task to occur. You might, perhaps, want to update a player's score when a bad guy has been destroyed in your game.

The if Statement

Making decisions in Lua, as with countless other programming languages, is performed using the 'if' statement, otherwise known as a conditional statement. The 'if' statement accepts a comparison expression in the statement body. Following the statement, the developer supplies a block of code, terminated with the word 'end', that is to be executed only when the conditional statement returns true.

```
if myNum < 1 then
	doSomething()
end
```

The comparison expression is the code that exists between 'if' and 'then'. Only when this expression returns True will the nested block execute. If the expression returns False, then the block of code is skipped.

Notice how the body of the executing block is indented. This isn't a necessity, but it is considered good form, as it makes the code far more readable.

As well as using Boolean comparisons in a conditional statement, it is also possible to simply use the keyword True as your comparison expression, which would force the block of code to execute regardless. However, this defeats the object of the conditional statement, and one may as well forgo the statement altogether.

Using else

So, you can now make decisions based on comparisons. However, what if you wanted one block of code to execute if an expression returned True and another if the expression returned False? Well, you could perform:

```
if myNum < 1 then
	doSomething()
end
if myNum >= 1 then
	doSomethingElse()
end
```

This is perfectly legal code. However, it's a little long winded and doesn't always read very clearly. Anyone following your code will need to scrutinize the comparison expressions in order to verify that they both cover all eventualities, and that person could be you three months down the line. So, instead of using separate statements, you can use a single 'if'

statement for the initial condition and use the 'else' keyword to handle all other conditions, like this:

```
if myNum < 1 then
        doSomething()
else
        doSomethingElse()
end
```

This way, you can be sure that code will execute whatever the comparison expression returns.

Nesting if Statements

So, you have a block of code that executes if the comparison expression returns True and a block if the condition returns False. However, very often, once you're sure a condition has not been met, you may need to provide further comparisons in order to provide more than a simple two option condition. For instance, I might want a function to be called if my variable is less than zero, another to be called if my variable is zero and yet another if it is greater than zero.

Performing this task with 'if' and 'else' is pretty straightforward, and I'm sure you will have already worked this out. I can simply nest a new 'if' statement inside my 'else' block, giving me an extra two possible outcomes instead of one:

```
if myNum < 0 then
        doSomething()
else
        if myNum == 0 then
                doSomethingElse()
        else
                if myNum > 0 then
                        doSomethingDifferent()
                end
        end
end
```

Now, as you can probably see, I could have performed the same task without the inner most nested 'if' statement, but it does raise an interesting point. What if I have lots and lots of conditions? I could be nesting all day long. Thankfully, though, it is possible to remove the nesting but keep all the conditional statements, by placing each 'if' immediately after the parent 'else' keyword:

```
if myNum < 0 then
        doSomething()
else if myNum == 0 then
        doSomethingElse()
else if myNum > 0 then
        doSomethingDifferent()
```

```
end
```

You might also notice that I've done away with two of the 'end' keywords. This is because Lua perceives an 'else if' to be a continuation of the parent 'if' statement rather than a separate 'if' statement, so the further use of 'end' is no longer necessary.

Loops

You've seen that programming is all about decisions, but it is also about iteration. By this, I mean repeating tasks until particular criteria have been met. In programming lingo, this is known as looping.

Looping is the act of repeating a task or group of tasks a set number of times or until a condition has been met. Normally, during the loop, slight variations in the code will occur that will affect the outcome of the program. The loop statement will usually keep track of one or more of these variations so that it can terminate when the initial condition is satisfied.

There are several ways to perform a loop. Each one useful for a different given scenario.

The for Loop

The 'for' loop is arguably the most common type of loop statement. It is used in circumstances where you know beforehand the quantity of the condition to be met before the loop has begun.

The signature of a for loop looks like this:

```
for [var] = [start value], [end value], [step amount] do
	-- block of code
end
```

Essentially, the loop will initialize the variable [var] with the value [start value]. It will then perform a loop, repeatedly, until [var] is equal to or greater than [end value]. The [step amount] is the value to increase the variable with each loop. You can leave out the [step amount] if you want, and the variable will adjust by one with each iteration.

For example, to perform a loop ten times, I could do:

```
for num = 1, 10 do
	print (num)
end
```

This would print the values 1 through 10. Similarly, if I wanted to print only even numbers, I could add a step value of 2, like this:

```
for num = 1, 10, 2 do
	print (num)
end
```

This would make the variable 'num' increase in value by 2 with each iteration.

It is also possible to get the loop to count backwards by specifying a higher number for the starting value over the end value:

```
for num = 10, 1 do
	print (num)
end
```

When using the 'for' loop, you can opt to use variables rather than literal numbers, in the opening condition. For example:

```
for num = startVar, endVar do
	print (num)
end
```

The 'for' loop will evaluate the condition only once, so if the variables used in the condition change, the overall number of loops will not:

```
startVar = 1
endVar = 10
for num = start, end do
	print (num)
	startVar = startVar + 5
	endVar = endVar + 3
end
```

The above example will loop ten times exactly.

The while Loop

The 'while' loop works much like the 'for' loop, except it is usually used when you are not sure of the number of iterations required before meeting the necessary condition.

While loops function like a combination of an 'if' statement and a 'for' loop. Similar to the 'for' loop, a 'while' loop will repeat a block of code until a condition is met. Nevertheless, like the 'if' statement, the opening expression works by evaluating a comparison expression and executing the proceeding block of code if the condition evaluates as True. For example:

```
myVar = 0
while myVar < 100 do
	print (myVar)
	myVar = myVar + 1
end
```

Unlike the 'for' loop, the condition is evaluated with each iteration. In our example, the 'while' loop will repeat, printing out the value of myVar each time. The value of myVar is

increased by 1 with each loop, so the condition is eventually evaluated as False, thus allowing the application to move beyond the loop.

It is up to the developer to adjust the code inside a 'while' loop so that the condition will eventually be satisfied. In some circumstances, it is possible to have an error in the loop logic so the condition is never satisfied and the loop continues indefinitely (or at least until the program is force quit).

The repeat Loop

The 'repeat' loop works in much the same way as the 'while' loop with the exception of a couple of useful caveats. The biggest of these is the order in which the conditional expression is evaluated. Unlike the 'while' loop, the 'repeat' loop executes its code block before testing for the condition. So while the 'while' loop may never actually run its code block (if its condition is initially evaluated as False), the 'repeat' loop will always execute at least once. For example:

```
repeat
	print ("Woohooo!")
until True
```

Here, the word 'Woohooo!' will print once before the loop is abandoned.

As the above example shows, another difference between 'while' and 'repeat' loops is that, although the 'while' loops condition will cause a loop to ensue when evaluated to True and cease when evaluated to False, the opposite is true of 'repeat' loops. For that reason, if the condition is evaluated to True, the 'repeat' loop will not repeat and if it is False, the loop will continue.

Using break

Although loops are set to terminate when the given condition is met, there are times when unforeseen circumstances need you to end a loop somewhere in the middle of its code block. At such times, you can use the keyword 'break'.

For instance, suppose I had a loop that was to execute five times and print out the current iteration, but at the same time, I needed to check that a second variable was greater than a given value and if it wasn't, exit the loop. Such a situation may look like this:

```
secondVar = 5
for num = 1, 5 do
	if secondVar < 3 then
		break;
	end
	print (num)
	secondVar = secondVar - 1
end
```

Here, the application would print the values 1 through 3, then halt. As the condition of the 'if' statement returns true, so the 'break' keyword is met which exits the loop.

When you use a 'break' statement, it must be placed at the end of a code block. Otherwise, not only will code following a 'break' keyword never execute but Lua will throw an error.

```
secondVar = 5
for num = 1, 5 do
	if secondVar < 3 then
		break;
		print ('I will never be reached')
	end
	print (num)
	secondVar = secondVar - 1
end
```

Now, if you tried to run this example, Lua will complain with:

```
'end' expected (to close 'if' at line 3) near 'print'
```

Custom Functions

Previously in this appendix, you saw how Lua provides functionality that can be applied to your data in the form of string and numeric functions. Like nearly every other language, Lua also provides a method to cater for your very own custom functions.

The purpose of functions is to provide for abstraction and reuse. The abstraction occurs because the calculations applied to your data can exist outside of the current flow of information. When the function invocation is reached, the logic flow moves to the function declaration where the calculations take place, before moving back to the point of abstraction where the program continues, using the newly calculated values.

The reuse is applicable because function execution exists in one place within your code. You may call this function a thousand times, but you will only ever need to write the code once.

Functions can provide functionality for all manner of tasks. They might move a player character, update an animation, score or game state, or even send data across the internet to a remote machine. As the developer, it is up to you to choose what your functions do and how the logic in your application will be divided among those functions.

Defining a Function

Functions work much like 'if' statements and loops (block statements); they provide a signature that needs to be met by the application and, when it is met, the proceeding block of code is executed. The primary difference between functions and block statements are that functions can be invoked anywhere in the code, while block statements are only executable in the context that they are written.

Functions adhere to the following general structure:

function [name]([[[param1], param2], ...])

Each function must have a name. This, like variables, can consist of underscores, letters and numbers, but must not start with a number. Following the name is a parenthesized group of parameters the function requires. Functions can have as many parameters as you need, though when defining functions, readability should always be considered. Having a function with ten parameters can feel very overwhelming when put to use and doesn't read very well.

A function parameter works just like a temporary variable. When calling the function, you place the values relative to the position of the associated parameter and the function definition will store that value into a variable with the parameter name. This variable then exists for the lifetime of the function.

```
function addAndPrint( num1, num2 )
        print (num1 + num2)
end

myNum1 = 12
myNum2 = 8

addAndPrint(myNum1, myNum2)
-- outputs 20
```

Returning Values from a Function

In many languages, functions exist in two forms: sub-procedures and functions. The differences between these exist only so much as functions return a value while sub-procedures do not. In Lua, the same is true. However, rather than define them as two different types of entities, Lua defines sub-procedures as those functions that do not contain a return statement, and actual functions as those that do. Beyond this, there is no real definition to describe them as such.

In our previous example, the function 'addAndPrint' didn't return anything; it merely applied functionality to the supplied parameters. I could test for a returned value by checking for the values type, however, as no value exists, Lua would throw an error:

```
print (type( addAndPrint( 1, 2 ) ))
-- outputs
--      bad argument #1 to 'type' (value expected)
--      stack traceback:
--              [C]: ?
--              [C]: in function 'type'
```

We can fix this by using the 'return' keyword and passing it a value to return.

```
function add( num1, num2 )
    result = num1 + num2
    return result
end

print (type( add( 1, 2 ) ))
-- outputs 'number'
```

Return statements can exist anywhere within a function and as many times as you need. Though, like the ‘break’ keyword, it only makes sense to provide a ‘return’ statement at the end of a code block, as any expressions that exist after a ‘return’ statement will not be evaluated. Lua even insists on this by raising an error if a ‘return’ statement is not found at the end of its parent code block. For example:

```
function add( num1, num2 )
    result = num1 + num2
    return result
    print ('I will never be reached')
end

print (type( add( 1, 2 ) ))
```

Will output the error:

```
'end' expected (to close 'function' at line 1) near 'print'
```

Returning Nothing

In the previous illustrations, we’ve seen that functions with no ‘return’ statement do not return anything, but you will sometimes want to return from a function early without passing a value. You can do this using the ‘return’ statement as before, but on its own. Doing so halts the execution of the function while the returned value will be the same as not using the ‘return’ keyword at all:

```
function returnNothing()
    return
end

print (type( returnNothing() ))
-- outputs
--      bad argument #1 to 'type' (value expected)
--      stack traceback:
--          [C]: ?
--          [C]: in function 'type'
```

Returning Multiple Values

As stated earlier in this appendix, one of the most powerful features of Lua is the ability to return more than one value from a function. In many languages, returning more than one value from a function is not possible without first creating an object to hold those values

and using the object as the transport mechanism to get those values from the function to the body of code that invoked it. Lua, however, provides an alternative using multiple assignment.

Multiple Assignment in Variable Definition

Multiple assignment is the act of assigning multiple values to multiple variables inside a single expression. In this instance, the term single expression denotes the use of only one unary operator. For example, to assign three values to three variables using a single expression, all I need to do is separate the variables and values using commas on either side of the operator, like this:

```
var1, var2, var3 = 1, 2, 3
print (var1)
-- outputs 1
print (var2)
-- outputs 2
print (var3)
-- outputs 3
```

Lua then places each value to the right of the operator into the correct variable on the left of the operator by location.

If the number of items to either side of the assignment operator is less than the items on the other side, then Lua discards those items that are in excess. In the case of less values, the variables that have no matching value are set to Nil, while in the case of less variables, the values with no matching variable location are discarded from the expression altogether.

Multiple Assignment from Function Return Values

Returning multiple values from functions works in the same way as variable assignment in that, when returning the values, each value is delimited by a comma. The variables receiving these returned values are then treated in the same way as declared variables in a multiple assignment.

```
function getABC()
	return “a”, “b”, “c”
end

var1, var2, var3 = getABC()
```

Here, the variables var1, var2 and var3 will contain the values “a”, “b” and “c” respectively.

If, on the other hand, I only required the first item from the returned list, I could simply do the this:

```
myVar = getABC()
```

As a result, myVar now contains “a”, while the rest of the return values are discarded.

Multiple Return Values as Function Parameters

Just as multiple return values can be used to assign multiple variables, they can also be used to substitute function parameter lists. The list of values will automatically be placed into the correct parameter occupying the same location in the list. For example:

```
function printABC( varA, varB, varC )
	print( varA )
	print( varB )
	print( varC )
end

printABC( getABC() )
-- outputs
--	a
--	b
--	c
```

This works great when using the getABC function as the only parameter to printABC, but the rules change when further values are used. For instance:

```
function printABC( varA, varB, varC, varD )
	print( varA )
	print( varB )
	print( varC )
	print( varD )
end

printABC( getABC(), "d" )
```

Looking at this code, it would be assumed that the output would be the letters 'a' through 'd'. However, this is not the case. Instead, the parameter varA is populated with 'a' as expected, while varB is populated with 'd' and the remaining parameters with Nil. The reason for this has to do with Lua's rules for dealing with what are known as value lists.

Value Lists

Until now, whenever a group of comma delimited variables have been used, you have been working with value lists. This not only includes those variables used in multiple assignments, but also lists of parameters in function signatures. In fact, anywhere variables are used is considered as a value list, even if there is only one variable.

Value lists come with a set of rules that affect which values get used where when building lists. The rules state that, when building a list, only the first value of any list included in its construction will be adopted, with the exception of the last item in the list, whereby all values are used. Now, when creating a list of single items, this is not noticeable, as each item will already be singular and thus all assigned variables will be included. However,

when building value lists from values returned from functions, values may be discarded. For example:

```
print( getABC(), getABC(), getABC() )
```

Here, as parameters are value lists, the 'print' function can accept a value list as its parameters. The function 'getABC' returns a value list as its return value, but when the output of each function call is combined into the new list, only the last function call will include all of its return values. The first two calls will only provide the first value in their returned value lists in the construction. Thus, the output of the above statement would be:

```
a	a	a	b	c
```

Summary

This appendix introduced you to the basic elements of the wonderful language, Lua. Following through the comprehensive examples, you have learned:

- Lua's standard types, including Nil, Booleans, numbers and strings
- Numeric operators and math functions
- String concatenation and string functions
- Comparison operators and expressions
- 'if' statements and conditions
- Loops, including 'for', 'while' and 'repeat' and breaking free from loops
- Functions and return values
- Value lists

In appendix B, we'll cover more advanced topics, including:

- Variable scope
- Tables as arrays and objects
- Closures
- Side effects

Appendix B

Advanced Lua

This appendix on advanced Lua programming was supplied by the great people at Ansca Mobile.

Lua
Advanced Topics

In the previous appendix, you looked at the basics needed to program using the Lua language. If you like, you could probably skip much of this section and still have enough knowledge to begin learning the Corona platform. However, Lua has many rich and useful features and it does pay to understand them well in order to get the best use out of the language.

In this appendix, we'll look at some of those features, as well as covering more detail of the topics covered in the previous appendix, enforcing what you know about the Lua language and how to make best use of it.

Understanding Variables

In the previous appendix, we looked at the various types of variables and how to use them. There are many rules to using variables that weren't discussed, but which play an important part in how your applications work. Lua offers a very simple set of rules for how variables are governed which, in many references, often seem more complicated than they really are. We'll be taking a look at these rules now, so that you can get the most out of programming for Corona.

Global and Local Variables

Until now, every variable created in the examples in these appendices have been a global variable. Global variables are those variables that exist "application wide". This means that any variable created, wherever it is created, can be accessed anywhere else in the application. For example:

```
function makeVar()
	myString = "This is a global string"
	print (myNum)
	-- outputs 22
end
```

```
myNum = 22
makeVar()
print (myString)
-- outputs This is a global string
```

For many experienced developers, this wouldn't be the expected outcome as one would expect scope to come into play. Indeed, creating all variables as global variables is bad form, as it can cause unforeseen issues. For example, what would happen if a variable of the same name is used inside a function as well as outside?

```
function addFive( param )
        num = param + 5
        return num
end

num = 22
result = addFive( num )
print( num, result )
```

The developer creating this code may assume that the variable num inside the function is created there, when in fact it is created externally. The num variable inside the function is the same variable. Thus, instead of the predicted print result:

```
22      27
```

The output is:

```
27      27
```

Believe it or not, this is a feature of Lua and one we'll be taking advantage of later in this appendix when we come to discuss closures. In the meantime, however, this is likely not what you want to achieve. Thankfully, Lua provides a keyword that forces scope to be applied to variables in the form of local.

The local keyword, when applied to the beginning of a new variable declaration, forces a variable to exist in the scope that it was declared. Don't worry if you do not understand scope at this point, as we'll be discussing that in a moment. Just know that, if I create a variable using the local keyword, my variable will be limited to the code block that it was created. If we update our previous example with this in mind:

```
function addFive( param )
        local num = param + 5
        return num
end

num = 22
result = addFive( num )
print( num, result )
```

The output from the print statement will be:

22 27

Understanding Scope

Scope is the means to contain values to a given code block. Whenever a code block is used, such as an if statement, for loop or function, an area of memory is reserved known as a stack. This stack is responsible for storing data that is local to the code block, such as the variables declared using the local keyword. Any data declared local to a given code block is not visible outside of that block. For example:

```
local myVar = 22
if true then
	local myVar = 33
end
print (myVar)
```

Here, the output of the print variable is 22, which is the value of the myVar variable in the stack where the print function is invoked. The myVar variable that was created inside the if statement ceases to exist when the if statement has ended and is sent to be garbage collected by the Lua virtual machine. This can be proven by removing the initial variable declaration, like so:

```
if true then
	local myVar = 33
end
print (myVar)
```

Here, the output of the print function invocation is nil, as there is no such variable available in the current stack. In order to perpetuate the value outside of the code block, the value of the variable within the block needs to be passed to a variable that does exist outside of the block, like this:

```
local myVar1 = 22
if true then
	local myVar2 = 33
	myVar1 = myVar2
end
print (myVar1)
-- outputs 33
```

So, variables are not upward values, meaning they cannot be accessed outside of the code block from whence they were declared, but what about downward? If we were to swap the access of the variables:

```
local myVar = 33
if true then
	print (myVar)
end
```

The value 33 will indeed be printed. Therefore, while values created inside code blocks cease to exist when the block has ended, those variables that exist in parent code blocks are visible within the nested child blocks.

When using values in nested code blocks, creating a local variable of the same name will mean that further use of that variable name will use the locally declared variable, not the parent declared variable:

```
local myVar = 100
if true then
	print ( myVar )
	-- outputs 100
	local myVar = 200
	print ( myVar )
	-- outputs 200
end
print ( myVar )
-- outputs 100
```

Here, the value of the parent variable will be printed when requested in the if statement code block. Then, once the variable of the same name is created locally, it will be used in all further references within the same block. Once that block ends, however, the original declaration is used.

Functions and Variable Scope

One thing that may come as a surprise to you is that, when declaring a function, you are in fact declaring a variable. As stated in the previous appendix, Function is a variable type. Ergo, when creating a function, I can choose to do so in the following manner:

```
myFunc = function( param1, param2 )
	-- do stuff
end
```

This is exactly the same as declaring it like this:

```
function myFunc( param1, param2 )
	-- do stuff
end
```

Although they appear different, the two declarations perform exactly the same task; namely, creating a variable called myFunc and populating it with the function definition and code block reference.

The parameters defined within a function signature are local to the function block. When Lua encounters function parameters, memory is reserved within the functions stack and the values of the parameters are stored in the stack. The parameters then point to these memory locations.

Just like other variables, functions can also be created as global or local variables. When creating a function as a local variable, this is otherwise known as a closure.

Closures

Closures are a very powerful feature of the Lua language in that, while many of your functions will exist as very rigid processes for manipulating your data, closures exist in an extremely dynamic form.

In a nutshell, closures are local functions. This means that they exist in a variable that itself is tied to a given stack. However, the benefit of closures comes in their ability to be able to bake into themselves variables that exist in the same stack. For example:

```
function buildClosure( num )
	local ret = function()
		print ( num )
	end
	return ret
end

test1 = buildClosure( 22 )
test2 = buildClosure( 100 )

print ( test1() )
-- outputs 22
print ( test2() )
-- outputs 100
```

Here, the function buildClosure creates a function and stores it into the variable ret. This function is then returned, much like any other value. When calling the buildClosure function, we store the result into a new variable, which we can then invoke as a function. However, what is different in the above example is that the closure makes use of a variable that exists outside of its code block; namely, the num parameter of the buildClosure function. This value is baked into the closure and becomes a variable local to the closures code block. Thus, when the closure is invoked, the value of the variable is remembered and used in the closure body.

Being able to make use of this feature means that the number of parameters needed in the closure signature can be greatly reduced. In addition, the purpose of the closure can be met through its signature while reducing the amount of code within the closure itself. For example, as a function, I may require a closure that accepts only one parameter that is obvious to the purpose of the function, while the varying conditions at the time of the closures construction may change that purpose slightly. The body of code that invokes the closure doesn’t need to know anything about these conditions; merely what value is required, if any, to allow the closure to execute.

Garbage Collection

When using local variables in Lua, space is reserved within the stack where the variable is declared. When the code block owning the stack is ended, the whole stack is marked for garbage collection. This means that the Lua virtual machine will pass over the memory used by the stack and, if it is marked for collection, release it so that it can be used by other stacks or indeed, other applications.

Lua handles this feature automatically, so the coder doesn't really need to worry about the process. That is, assuming global variables and functions are not used.

Global variables and functions are not stored in a code block stack but instead exist in a special table reserved specifically for global objects. When using a global variable, it is up to the coder to destroy the variable when it is no longer needed. This is done by setting the variable to the nil value. The same is also true when using global functions:

```
function myGlobalFunc()
	print ( "I'm in a global function" )
end

myGlobalFunc()
-- outputs I'm in a global function
myGlobalFunc = Nil
myGlobalFunc()
-- results in the error - attempt to call global 'myGlobalFunc' (a nil value)
```

This will become ever more important when using variables in loops and assigning objects to them. If each object exists in a global variable and you fail to set them to Nil when you are done with them, they will gradually build up in memory and cause your applications to slow down considerably. Remember, mobile devices have limited memory, so you must be sure to free up as much memory as possible whenever the opportunity arises.

Alternatively, it is perfectly feasible to set all variables and functions as local values, thus, removing this requirement altogether. However, this becomes more of a trade as you will then have more concern with regard to scope and what variables can be accessed where.

Functions with Variable Arguments

As you've seen previously, functions can have as many arguments as you like. The arguments of a function are value lists, which means Lua treats those arguments as a group of values, just as you can pass groups of values as return values from functions or as values in assignments to groups of variables.

When writing functions, you'll normally know exactly what values are necessary for the function to carry out its purpose. However, there will also be times when you don't know the number of arguments, or when the number of arguments passed to a function change its purpose. For example:

```
function addValues( arg1, arg2, arg3, arg4, arg5 )
```

```
        if arg1 == Nil or arg2 == Nil then
                return
        end
        local ret = arg1 + arg2
        if arg3 ~= Nil then
                ret = ret + arg3
        end
        if arg4 ~= Nil then
                ret = ret + arg4
        end
        if arg5 ~= Nil then
                ret = ret + arg5
        end
        return ret
end

print ( addValues( 5 ) )
-- outputs nothing
print ( addValues( 5, 4 ) )
-- outputs 9
print ( addValues( 5, 4, 3 ) )
-- outputs 12
print ( addValues( 5, 4, 3, 2, 1 ) )
-- outputs 15
print ( addValues( 5, 4, 3, 2, 1, 100 ) )
-- outputs 15
```

Here, we want to have a function that outputs the sum of any numbers passed to it. We've catered for up to five possible arguments, but if more are passed, then only the first five are calculated, leaving further arguments to be discarded.

What if we want to support an infinite number of arguments? It wouldn't be feasible to enter conditions for every possible condition, as that would be unruly. So, how do we resolve this problem? Well, we could request that arguments are passed in as an array, which will be explained later, but that would be impractical, as it would mean the user will need to create the array before calling the function. Instead, the answer is to use the vararg (variable arguments) operator '...'.

The VarArg Operator

The vararg operator represents a value list in a simpler format. To use it, you replace the arguments variables in a function signature with the operator:

```
function addValues( ... )
        local arg1, arg2, arg3, arg4, arg5 = ...
        return arg1 + arg2 + arg3 + arg4 + arg5
end

print ( addValues( 5, 4, 3, 2, 1 ) )
-- outputs 15
```

You could also use the vararg operator to represent a partial number of the expected arguments. Thus, if you knew you'd always have at least two arguments, we could rewrite the function:

```
function addValues( arg1, arg2, ... )
    local arg3, arg4, arg5 = ...
    return arg1 + arg2 + arg3 + arg4 + arg5
end
```

As you can see, the operator becomes a representation of the value list, so you only need to use it in the function signature once. The operator can then be used as the assignment values or wherever value lists can be used. Now, looking at the above example, it doesn't look much like it's helped, as the variables the arguments would have facilitated still need to be created. However, their value comes in to play when we start using some choice functions provided just for value lists.

Select

The select function accepts a value list, like the vararg operator, and returns a requested number of items. The number of items to return forms the first argument, while a value list provides the second argument, like this:

```
print ( select( 3, "a", "b", "c" ) )
-- outputs      a       b       c
print ( select( 1, "a", "b", "c" ) )
-- outputs      a
```

Now, this might not seem very useful, but it becomes extremely valuable if we exchange the first argument for the length operator as a string:

```
print ( select( "#", "a", "b", "c" ) )
-- outputs 3
print ( select( "#", 1, 2, 3, 4, 100 ) )
-- outputs 5
```

As you may have guessed, the length operator, in string format, forces select to return the number of items in the value list. Thus, as with our previous vararg conundrum, we could use select to provide the bounds of a loop in order to iterate through the items in a value list, like this:

```
function addValues( ... )
    local args = {...}
    local ret = 0
    for i = 1, select( "#", ... ) do
        ret = ret + args[i]
    end
    return ret
end

print ( addValues( 5, 4, 3, 2, 1, 100 ) )
```

-- outputs 115

In this example, to get access to the arguments within the loop, we've had to convert our value list into a table. Don't worry about this too much, as we've taken a bit of a jump. We'll be looking at tables in a little while.

Recursion

Recursion is the means for a function to call itself. This might seem an odd thing to do, but it is extremely powerful. In fact, recursion is actually the foundation of some languages and makes easy work of processing data, which is surely the point of all applications. For example, there would be a lot of recursion present in the parser that converts your Lua scripts into executable objects within the virtual machine.

Performing recursion is simple. You simply invoke a function within itself:

```
function recursive()
	print( "I'm recurring" )
	recursive()
end
```

Now, this is just an example to outline how to recursively call a function, but I certainly wouldn't recommend trying this exact example. The problem here is that the function would repeat indefinitely, being called over and over again. The likelihood is you won't even see any output as Lua wouldn't have the chance to perform any actual action on the print invocation.

To get around this problem, all recursive functions need a condition. You would only want your function to recur if the condition has or hasn't been met, and to cease recurring if the opposite is true. For example:

```
function recursive( counter )
	counter = counter + 1
	print( "I'm recurring " .. counter )
	if counter < 5 then
		recursive( counter )
	end
end

recursive( 0 )
-- outputs
--	I'm recurring 1
--	I'm recurring 2
--	I'm recurring 3
--	I'm recurring 4
--	I'm recurring 5
```

The value for the condition needs to be passed with every call to recursion, so that you can evaluate the new value and see if the function should continue recurring. The value is local to the currently recurring function.

Our above example is very similar to a for loop. We've supplied a start value and a condition, so the loop occurs so long as the condition is true. The difference between a recursive call and a for loop comes from the ability to continue a recurring function call after the next iteration has been invoked. Thus, while we can do the above example using a for loop, we couldn't very well do the following:

```
function recursive( counter )
	counter = counter + 1
	print( "I'm recurring " .. counter )
	if counter < 5 then
		recursive( counter )
	end
	print( "Too late, I recurred, already! I was recursion " .. counter )
end

recursive( 0 )
-- outputs
--	I'm recurring 1
--	I'm recurring 2
--	I'm recurring 3
--	I'm recurring 4
--	I'm recurring 5
--	Too late, I recurred, already! I was recursion 5
--	Too late, I recurred, already! I was recursion 4
--	Too late, I recurred, already! I was recursion 3
--	Too late, I recurred, already! I was recursion 2
--	Too late, I recurred, already! I was recursion 1
```

As you can see, while each function call invokes itself, this leads to the function call stack nesting itself with each invocation. Then, once all calls are complete, the nested calls finish their execution in the reverse order.

The Table Type

In Appendix A, we noted the six types of variable that you will be using to build your applications. So far, we've managed to cover five of those types. The sixth type, Table, is a more complex type.

Tables are very flexible and malleable objects. Like putty, you mold them and shape them, and when you're done with them you can mold them into something else. They are the only data structure in the Lua language. Any other data structure you may encounter in your Lua career will be structures that use tables at their base.

If you're a veteran coder, you might like to think of tables as arrays, associative arrays, anonymous objects, and class instances all rolled into one crazy little tool. This might seem

funky in the extreme and maybe a little worrisome, in some respects. But I'm sure that once you've used them a little while, I'm sure you'll agree to their usefulness.

Associativity

Lua's tables work by associativity. You might like to think of them as the opposite of value lists. While value lists are loosely coupled groups of variables with no name stored in a common structure, tables are relatively tightly coupled groups of variables, each named and stored in a common structure.

To create a table, you use the curly braces { and }. Then, each property of that table is listed, comma delimited, as a series of variable assignments:

```
local myTable = { ["a"] = 5, ["b"] = 4, ["c"] = 3, ["d"] = 2, ["e"] = 1 }
print ( myTable["a"], myTable["d"] )
```

Each of the variable assignments are known as key - value pairs, where the value to the left of the assignment operator is the key and the value to the right of the assignment operator is the value. As you can see, the key in our example is denoted by a string contained in the square brackets [and]. The value for the key can be anything, so long as it's not equal to Nil, thus, the following is also legal:

```
local functionKey = function()
        print("I'm a key")
end
local t = { ["I'm a key"] = 5, [-24.7] = 4, [true] = 3, [false] = 2, [functionKey] = 1 }

print ( t["I'm a key"] )
-- outputs 5
print ( t[-24.7] )
-- outputs 4
print ( t[functionKey] )
-- outputs 1
print ( t[true] )
-- outputs 3
print ( t[false] )
-- outputs 2
```

When choosing names for your keys, if you choose to use a string that fits the rules for variable naming, then you can omit the quotes and square brackets altogether. Likewise, the same applies when accessing their values. Thus, the following examples are equivalent to one another:

```
local myTable = { one = 1, two = 2, three = 3 }
print ( myTable["one"], myTable["two"], myTable["three"] );
-- outputs     1     2     3

local myTable = { ["one"] = 1, ["two"] = 2, ["three"] = 3 }
print ( myTable.one, myTable.two, myTable.three );
-- outputs     1     2     3
```

```
local myTable = { one = 1, two = 2, three = 3 }
print ( myTable.one, myTable.two, myTable.three );
-- outputs     1     2     3
```

As you can see, omitting the quotes and square brackets when accessing a value requires the inclusion of the '.' (period) operator, known as the index operator, to separate the table name from the key. The reason for this is simply that, by omitting the index operator, the table name and key would appear to the Lua interpreter as a new variable name, such as myTableone or myTabletwo. You cannot, however, use the index operator when including the quotes and square brackets.

Similarly, it is not possible to include the square brackets but omit the quotes, like this:

```
print ( myTable[one] )
-- outputs nil
```

To do so would be akin to passing a variable as the key identifier. You can, however, use the above notation if the variable one holds a value equal to a key name:

```
local myTable = { "I'm a key" = 1 }
myKey = "I'm a key"
print ( myTable[myKey] )
-- outputs 1
```

Tables as Arrays

So, we've seen that any value can be used as a key, but Lua pays special attention to tables created using integers as keys.

In other languages, a structure containing data stored using positive integers as keys is called an array. As the keys can be perceived as linear to one another, these languages often also provide functions for dealing with the structures data in a linear fashion. Lua is no exception and provides such functions for working with tables that have keys that are integers.

Array Indices

Integer keys are otherwise known as indices. Thus, an item of data in an array is stored by index. This is not to be confused with the index operator, which has a related but different purpose. The indices in a Lua table can be any value from 1 and above, but should be in linear order.

Creating Arrays

Arrays are created much in the same way as tables are created. The following is a perfectly acceptable array instantiation.

```
local myArray = { [1] = "a", [2] = "b", [3] = "c" }
print ( myArray[2] )
-- outputs b
```

The following is also acceptable:

```
local myArray = { [8] = “a”, [9] = “b”, [10] = “c” }
print ( myArray[10] )
-- outputs c
```

Now, this is nothing new. We’ve already seen just such a constructor in previous examples and these form perfectly acceptable definitions for tables, too. However, while the construction of tables assume that keys are specified, arrays do not require keys to be initially assigned to values. For example:

```
local myArray = { “a”, “b”, “c” }
print ( myArray[2] )
-- outputs b
```

Here, we refrained from specifying keys, so the Lua interpreter provided them for us. The first value Lua assigns as a key is 1, while each following key assigned is incremented by 1. So, the keys for the previous example would be 1, 2 and 3.

Arrays are Tables Too!

As was previously mentioned, arrays are a type of table. However, what hasn’t been stated is that, while tables can be arrays, making it so doesn’t stop it from being a table. Thus, I can quite happily provide non-array like key-value pairs and still have it behave as an array (or table) when the need arises. For example:

```
local myConfusedArray = { [1] = “a”, [2] = “b”, three = “c” }
print ( myConfusedArray.three )
-- outputs c
print ( myConfusedArray[2] )
-- outputs b
```

Also, as noted earlier, we can still insist on letting Lua infer our indices for us, even if we have non-integer keys, like this:

```
local myConfusedArray = { “a”, “b”, three = “c” }
print ( myConfusedArray[1] )
-- outputs a
print ( myConfusedArray[2] )
-- outputs b
print ( myConfusedArray.three )
-- outputs c
```

The keys inferred by the interpreter are still added in an incremental fashion, but once inferred, the table is expanded to include a different value using a named key. This value is not considered part of the array, even if it is part of the table. Lua creates an array from any given group of variables passed with a linear set of indices at the beginning of the table

constructor. Breaking that linearity or starting the array anywhere but at the beginning of the constructor, ends the array construction. For example:

```
local myConfusedArray = { “a”, “b”, three = “c”, “d” }
print ( myConfusedArray[1] )
-- outputs a
print ( myConfusedArray[2] )
-- outputs b
print ( myConfusedArray.three )
-- outputs c
print ( myConfusedArray[3] )
-- outputs nil
```

Here, it might be assumed that the value “d” would be given an inferred index, but it doesn’t. In fact, it doesn’t receive a key at all, so is not accessible. Also:

```
local myArray = { [1] = "a", [2] = "b", [3] = "c", [5] = "d" }
```

Here, items 1 through 3 are part of the array, while item 5 is considered a table item. This will become important when using the array specific functions, which we’ll see in a moment.

Specifying indices in an array doesn’t need to exist within the constructor in a linear fashion so long as the specified indices themselves are linear:

local myArray = { “a”, “b”, [4] = “d”, tableKey = “some value”, [3] = “c” }

The items “a” through “d” are all part of the array, while “some value” is a table item.

Unpacking Arrays

So far, we’ve seen how to create arrays from value lists using the curly braces { and }. It is also possible to convert an array back into a value list using the unpack function, like this:

```
local myTable = { "1", "2", "3" }
item1, item2, item3 = unpack ( myTable )
print ( item2 )
-- outputs 2
```

The unpack function is particularly useful when you want to pass an array of items as parameters to a function. For example:

```
local myTbl = { "1", "2", "3" }
printArgs ( unpack( myTbl ) )
-- outputs	1	2	3
```

Finding the Length of an Array

Lua allows for finding the number of items in the array part of a table, using the length operator '#'. We looked at the length operator and how to use it when looking at strings in the previous appendix. The operator is used in the same fashion with Lua arrays, like this:

```
local myArray = { "a", "b", "c" }
print (#myArray)
-- outputs 3
```

Looping Over Arrays with ipairs

Using the length operator, it is now possible to create a numeric loop to iterate over arrays:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", "f", "o", "x" }
    local str = ""
    for i = 1, #myArr do
        str = str .. ", " .. i .. " = " .. myArr[i]
    end
    print ( str )
    -- outputs , r, e, d, l, a, z, y, f, o, x
```

Now, this looks great, but what would happen if the starting index of the array is not 1? As there's no way to check the starting index, one would have to perform a check against Nil for each item iterated so as not to cause any runtime errors. This could be quite an inefficient process if the starting index is a large number.

To avoid this problem, Lua provides the simple function ipairs, which performs the same task as the length operator, with the exception that it returns each index and associated value of the array. The ipairs array accepts the table object as its only parameter:

```
    local myArr = { "r", "e", "d", "l", "a", "z", "y", "f", "o", "x" }
    local str = ""
    for indx, val in ipairs( myArr ) do
        str = str .. ", " .. indx .. " = " .. val
    end
    print ( str )
    -- outputs , 1 = r, 2 = e, 3 = d, 4 = l, 5 = a, 6 = z, 7 = y, 8 = f, 9 = o, 10 = x
```

Adding Values to Arrays

Once an array has been constructed, there are two ways to add an item to an array. The first option is to find the length of the array, then add an item to the next slot. For example:

```
    local myArray = { "a", "b", "c", [99] = "d", [4] = "e" }
    print (#myArray)
    -- outputs 4
    myArray[#myArray+1] = "d"
    print (#myArray)
    -- outputs 5
```

This works fine, but it a little confusing to look at. The value myArray is used twice in the same expression and it's not totally obvious as to what is happening. An easier way to do this is to use the insert function.

Insert is one of Lua's many table functions. Table functions belong to the table object, much like the string functions from the previous appendix belonged to the string object. To use insert, you need to supply the array to add the value to and the item to insert as arguments:

```
local myArray = { "a", "b", "c", [99] = "d", [4] = "e" }
print (#myArray)
-- outputs 4
table.insert ( myArray, "d" )
print (#myArray)
-- outputs 5
```

The output from insert is exactly the same as the previous example, where the value to be added is appended to the higher end of the array, but proves a lot easier to follow when reading back through your code.

Removing Values from Arrays

Just as table provides a function for adding an item to an array, it also provides a function to remove an item from an array. The remove function doesn't need to know the value to be removed; it simply removes whatever is at the highest index. Thus, remove accepts a single argument: the array from which the item should be deducted. For example:

```
local myArray = { "a", "b", "c", [99] = "d", [4] = "e" }
print (#myArray)
-- outputs 4
table.remove ( myArray )
print (#myArray)
-- outputs 3
```

The remove function also returns the value of the item it's removing, so it can be stored in a separate variable if needed:

```
local myArray = { "a", "b", "c", [99] = "d", [4] = "e" }
print (#myArray)
-- outputs 4
local val = table.remove ( myArray )
print (val)
-- outputs e
```

Converting Arrays to Strings

Sometimes, it may be necessary to convert an array into a delimited string. The table object provides the concat (short for concatenate) function for this task. The concat function accepts the array to convert as its first argument and the symbol to use within the

concatenation as the second argument. The function returns the newly formatting string version of the array for you to use as you like. For example:

```
local myArr = { "a", "b", "c" }
local str = table.concat ( myArr, ", " )
print ( str )
-- outputs a, b, c
```

As you can see, the symbol to concatenate with can be any number of characters and whitespaces, leaving the returned string easier to read, if that's what you want. You can also leave out the delimiter argument, which is the same as passing nil or an empty string. Thus, doing so outputs all items with no delimiter:

```
local myArr = { "a", "b", "c" }
local str = table.concat ( myArr )
print ( str )
-- outputs abc
```

The concat function can also accept two more arguments which represent the starting item in the array and the ending item in the array, respectively, with which to construct the concatenated string:

```
local myArr = { "a", "b", "c", "d", "e", "f" }
local startItem = 2
local endItem = ( #myArr ) - 1
local str = table.concat ( myArr, ", ", startItem, endItem )
print ( str )
-- outputs b, c, d, e
```

Sorting Arrays

When working with arrays, it is common to want to sort the values contained therein by a given formula. For example, you may want to rearrange the values so that they descend in a linear fashion. The table object provides the sort function for making this possible.

Note that the sort function only works with array items and not general table items. This is because the items in an array are not bound to their indices, while table key-value pairs are tightly bound.

To use sort, you need to pass it the array to sort and a closure which will perform the actual sort. The sort function applies the closure to all items within the array until all items are sorted:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", "f", "o", "x" }
local isSmaller = function( a, b )
	return ( a < b )
end
table.sort ( myArr, isSmaller )
print ( unpack( myArr ) )
-- outputs	a	d	e	f	l	o	r	x	y	z
```

The closure passed to sort needs to accept exactly two arguments and compute a condition. If the closure returns Nil or False, the condition is considered to have calculated as false, while any other value denotes a matched condition.

Finding the Largest Index

We've already seen how an array can be queried for its size, but we've also seen how brittle the Lua array can be. By having holes in an array, it is possible to lose chunks of data otherwise tied to an array, and this can cause problems. For example:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", [13] = "f", [14] = "o", [15] = "x" }
print ( #myArr )
-- outputs 7
```

Here, although we have indices with values as high as 15, the break occurs after the first seven items, thus 7 is returned as the length of the array. This is fine when you want to know what items will be affected by array functions, but it is not acceptable when you rely on the indexed data within a table and you have no idea as to the largest index; it may be in the thousands, making any chance of looping through the data as inefficient or impossible, even when using the ipairs function. Thankfully, the Lua table object provides the maxn (or maximum number) function to help resolve this problem.

The maxn function accepts a table as its only argument and returns its highest used index, whether or not that index forms part of an array:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", [13] = "f", [14] = "o", [15] = "x" }
print ( table.maxn( myArr ) )
-- outputs 15
```

Despite the break in the array constructor, Lua was able to return the highest used index, which we could then use inside a loop in order to perform calculations on all items, array or not:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", [13] = "f", [14] = "o", [15] = "x" }
local str = ""
for i = 1, table.maxn( myArr ) do
        if myArr[i] ~= Nil then
                str = str .. ", " .. myArr[i]
        end
end
print ( str )
-- outputs , r, e, d, l, a, z, y, f, o, x
```

One very good use for this function, which is rather commonly performed, is to find all indexed items within a table, including those that are not affected by array functions, in order to create a new, valid array. For example:

```
local myArr = { "r", "e", "d", "l", "a", "z", "y", [13] = "f", [14] = "o", [15] = "x" }
```

```
local newArr = {}
for i = 1, table.maxn( myArr ) do
	if myArr[i] ~= Nil then
		table.insert( newArr, myArr[i] )
	end
end
print ( #newArr )
-- outputs 10
```

After the loop, the array newArr would contain all of the values from myArr, in the same order and without any breaks. This would make all indexed items valid when array functions are applied.

More on Tables

So, we've seen a little bit about tables and how they can be used as arrays, but what about their use as, well, plain old tables? How can storing data in key-value pairs, when they cannot be subjected to array functions, be useful?

The answer is two-fold. First, Lua provides a couple of functions in its arsenal for non-indexed items held in tables, which provide usefulness to your key-value pairs. We'll be looking at those next.

Secondly, as stated previously, tables are Lua's only complex data structure. Thus, they are the only route to Object Oriented Programming (OOP) available to Lua developers. Now, Lua is not strictly an object oriented language. It is actually a procedural language with a level of flexibility that allows for object oriented and functional features. To this extent, one could choose to develop in Lua using almost pure procedural, object, or functional paradigms.

Object oriented programming in Lua will be discussed a little later in this section.

Iterating Through Table Keys

We've seen how to iterate over tables using a numeric for loop, but what about any other keys that aren't integers? Well, to accomplish this task, Lua has provided a couple of options.

The next Function

The next function accepts a table as its first parameter and a key, whether numeric or not, as its second parameter. The function then returns the next available key in the tables list of keys. If the second parameter is omitted or the value nil is passed, the first key in the list is returned. Likewise, if the last key in the list is passed, then nil is returned:

```
local myTbl = { ["I'm a key"] = 5, [-24.7] = 4, [true] = 3, lastIndx = 2 }
local str = ""
repeat
	indx = next ( myTbl, indx )
	if indx ~= Nil then
		str = str .. ", " .. myTbl[indx]
```

```
        end
until not indx
print ( str )
-- outputs , 2, 4, 3, 5
```

The list of keys are returned by next in a seemingly random order. Therefore, next is usually only useful when iterating over the entire table.

The pairs Function

We previously looked at using the ipairs function for iterating over integer keys. Well, Lua provides an identical function for working with all keys in a table, called pairs. While the ipairs function returned each index in an array in a numeric order, the pairs function works more like the next function, in so much as each key is returned seemingly random. For example:

```
local myTbl = { ["I'm a key"] = 5, [-24.7] = 4, [true] = 3, lastIndx = 2 }
local str = ""
for indx, val in pairs ( myTbl )
        str = str .. ", " .. val
end
print ( str )
-- outputs , 2, 4, 3, 5
```

The differences between next and pairs are extremely negligible, so their use is often based on preference. However, as a rule of thumb, pairs would normally be used with priority to next due to its syntactical elegance, while next would be used when the need arises that an iteration over a table will need to resume beyond the beginning of the list of table keys.

Object Oriented Programming in Lua

Object oriented programming is a useful paradigm for many reasons. Primarily, OO programming is a means to map code to real world objects. This has the effect of simplifying the overall model of code, making it easier to read. The other benefits of OO programming include code reuse, autonomy, and the ability to make projects more friendly in team environments.

Unfortunately, the ongoing debates, best practices and patterns associated with OO programming are out of the scope of this book. Indeed, OO programming has spawned the production of tomes dedicated to nothing but this topic. However, there are several great books in the Apress library that target OO programming with specific languages, and the internet is always a great resource for this subject. In the meantime, we'll take a brief look at OO programming in Lua.

Creating an Object

Objects in Lua are possible thanks to the flexibility of tables, the nature of closures, and the ability to override Lua operators using, what are known as, metamethods.

Starting at the beginning; to create an object, we simply create a new table:

```
local myTable = {}
```

In every sense of the word, myTable is an object. It can have properties in the form of key - value pairs containing literal values or other tables, and it can have methods (which is OO talk for object based functions). The way in which Lua differs from other OO languages, with the exception that Lua is not an OO language itself, is with regard to the key fact that Lua doesn't have classes. Ergo, any new object created in Lua is a living entity capable of being entirely different to any other object, even if they start off the same. For instance, unlike any object oriented language, it is very possible to create two objects the same, then to modify them at runtime so that neither object even remotely resembles the other.

Designing Objects

As stated previously, objects are simply tables that contain values and functions. The idea is that the functions contained in the object can, and often do, manipulate the values within the same object. As these objects map to real paradigms, they will normally facilitate a specific group of tasks. For example, should I want to develop a game about a dog, I might need a dog object. This object may have methods that allow the dog to bark, run or wag its tail, while the dog's properties might describe the direction the dog is running, whether its tail is currently wagging and what the dog might say when it barks. The following is a simple object that shows this example in action:

```
directions = { north = "northerly", east = "easterly", south = "southerly", west =
        "westerly" }
dog = {}
dog.barkMessage = "I'm hungry. Feed me!"
dog.direction = directions.north
dog.isWagging = false
dog.bark = function()
        print( dog.barkMessage )
end
dog.doWag = function()
        dog.isWagging = true
end
dog.stopWag = function()
        dog.isWagging = false
end
dog.run = function()
        print ( "Dog is running in the " .. dog.direction .. " direction" )
end

dog.bark()
-- outputs I'm hungry. Feed me!
dog.doWag()
print ( dog.isWagging )
-- outputs true
dog.stopWag()
print ( dog.isWagging )
-- outputs false
```

```
dog.run()
-- outputs Dog is running in the northerly direction
dog.direction = directions.south
dog.run()
-- outputs Dog is running in the southerly direction
```

This is a start, but there are lots of inefficiencies here. To begin with, objects are supposed to be self-contained (or 'encapsulated'). However, here, in order to access values of the dog object within the objects methods, we need to relate specifically to the dog object. This is poor practice and could lead to problems further down the line. For example, what happens when we want to create more than one of these objects? We can't have all instances of this object accessing the same data.

The self Property

To combat this, Lua provides a key, which is attached to all table instances, called self. This key, or property as it is known in OO speak, returns the current instance of the table used in an expressions context. In other words, if I have two tables, each with a copy of a given function, if the function references the self property, it references the table making the call to the function. For example:

```
dogOne = { name = "Fido" }
dogOne.bark = function( self )
        print ( self.name .. " is barking" )
end
dogTwo = { name = "Colin" }
dogTwo.bark = function( self )
        print ( self.name .. " is barking" )
end
print ( dogOne.bark( dogOne ) )
-- outputs Fido is barking
print ( dogTwo.bark( dogTwo ) )
-- outputs Colin is barking
```

Okay, so, this example may be raising some eyebrows right now. I mean, this so-called special property, **self**, is nothing more than an argument passed from the function invocation. Of course the correct object will be passed to the function body; we're passing it explicitly.

Now, that may be obvious, but we're not quite finished. The issue we have here is in the way the method was called. In the above example, we've called the method as though it were a function of a table rather than a method of an object. The minor difference is in the syntax. While in a table function call, we use the '.' (period) operator, Lua provides a nifty new operator just for object method calls; the ':' method operator.

The method operator doesn't really have a name (besides colon), so if anyone asks, you saw it here first.

The purpose of the method operator is to remove the need to specify self in the function declaration or the current object when invoking a method. For example, we could quite happily rewrite the above example like this:

```
dogOne = { name = "Fido" }
function dogOne:bark()
    print ( self.name .. " is barking" )
end
dogTwo = { name = "Colin" }
function dogTwo:bark()
    print ( self.name .. " is barking" )
end
print ( dogOne:bark() )
-- outputs Fido is barking
print ( dogTwo:bark() )
-- outputs Colin is barking
```

In essence, both forms of calling the function amount to the same thing. We're doing nothing new besides using a new way of writing the same task. Lua reads both types of syntax to mean the same thing. Thus, the following are exactly the same to Lua:

```
myObject.doFunc( myObject )
myObject:doFunc()
```

Likewise, the following are also equal:

```
myObject.doFunc = function( self ) print( self ) end
function = myObject:doFunc() print( self ) end
```

In the latter example, Lua understands that the self keyword may be used within the function, so sets it as the first parameter and hides it. Then, when calling the function, if using the method operator, Lua automatically passes the correct table instance to the function as the first parameter. Clever, huh?

Metamethods

We've covered a lot so far. You should now be able to see how objects can be used in your code and how you might design them to make your code more efficient. However, we're still lacking some important OO features. For example, how would we create multiple instances of an object without building each one explicitly? How would we deal with requests to properties that do not exist or execute methods when particular properties are queried? How would we perform polymorphism (the means to extend one object on top of the functionality of another)? Each of these features are possible in Lua, to a degree, but in order to get to those features, we need to look at what are otherwise known as metamethods.

Understanding Metamethods

Since the beginning of the previous appendix, we have been using various functions in Lua without realizing we've been using them. If you consider any process Lua performs,

whether it is performing a mathematical equation, concatenating strings, or accessing a value in a table, Lua is executing its own functions to achieve them.

The good news, for us, is that Lua also provides a means to hijack these processes to provide our own functionality, which include processes with otherwise incompatible data types. The functions we provide to override these processes are called metamethods.

Metamethods are just like any other methods. There is no specific syntactical requirement beyond that the function created to override a given process needs to follow a strict function signature. Once created, the function then needs to be registered to a parent table object.

Registering Metamethods with setmetatable

To register metamethods, each function first needs to be added to a metatable. Metatables are just like any other table, with the exception that it contains functions used as metamethods.

Once the metatable has been constructed, it is bound to a new table instance using the setmetatable function. setmetatable registers each function in the metatable with the passed table instance. For example:

```
local tbl = { value = 100 }
local metaAdd = function( x, y )
        return { value = x.value + y.value }
end
local metaTbl = { __add = metaAdd }
setmetatable ( tbl, metaTbl )

newTbl = tbl + tbl
print ( newTbl.value )
-- outputs 200
```

Here, the addition operator is overridden by the __add metamethod. The method adds the values of the value property and creates a new object which it promptly returns. The purpose of using a metatable to store the metamethods before registering the functions are so that the metamethods can be registered with further objects.

Operator Metamethods

Most of the operators supplied by Lua have a matching metamethod that can be used to override the operator when used with tables. You've already seen the addition metamethod in use. The following table lists the other operator metamethods available:

Operator	Metamethod Signature	Description
+	__add(a, b)	Called when a value is added to the parent table using the + operator
-	__sub(a, b)	Called when a value is subtracted from the parent table using the - operator
*	__mul(a, b)	Called when the parent table is multiplied by a value using the * operator
/	__div(a, b)	Called when the parent table is divided by a value using the / operator
%	__mod(a, b)	Called when a value is used as a modulus of the parent table using the % operator
^	__pow(a, b)	Called when the parent table is raised to the power of a value using the ^ operator
-	__unm(a)	Called when the parent table is used with the unary operator -
..	__concat(a, b)	Called when a value is concatenated to the parent table with the .. operator
#	__len(a)	Called when the length of the parent table is queried with the # operator
==	__eq(a, b)	Called when the parent table is compared with a table also registered with the same metamethod
<	__lt(a, b)	Called when the parent table is compared to a value using the < operator
<=	__le(a, b)	Called when the parent table is compared to a value using the <= operator

The parameters associated with a metamethod signature coincide with the signature for the corresponding operator. Thus, when adding, the expression:

c = a + b

is handled by the __add metamethod in the following way:

c = __add(a, b)

Where an operator metamethod accepts two parameters, the first parameter always refers to the value to the left of the operator while the second parameter refers to the value to the right. Each of the operator metamethods can receive values of any data types, with the exception of the equality operator (==), which insists that the values on either side of the operator must be registered with the same __eq metamethod.

Accessing Values with the __index Metamethod

The __index metamethod overrides the '.' period operator used when accessing a property of a table for its value. The novel feature of __index is that, when querying a property, it doesn't actually need to exist. This facilitates a lot of handy tricks useful in OO programming.

Suppose we wish to access a property but have a function execute when this happens? This is a task commonly provided by getters in many OO languages. The trick is, rather than create the property, we instead provide a function for the __index metamethod that checks for the property to be requested. Then, when it is, execute the necessary functions and return a custom value. For example:

```
Object = { value = 0 }
mt = {}

function Object:add( num )
	self.value = self.value + num
	return self.value
end

function Object:sub( num )
	self.value = self.value - num
	return self.value
end

mt.__index = function( tbl, key )
	local ret = tbl.value
	if key == "addFive" then
		ret = tbl:add( 5 )
	elseif key == "subThree" then
		ret = tbl:sub( 3 )
	end
	return ret
end

setmetatable( Object, mt )

print ( Object.addFive )
-- outputs 5
print ( Object.subThree )
-- outputs 2
```

Here, we had no need to provide actual properties for the object, but instead resolved calls to pseudo properties in order to generate dynamic output.

Another use for the __index metamethod is to apply methods from one object to another object. This is akin to facilitating class like functionality, whereby a class defines the available functions for a type of object, while an object instance implements the functions themselves:

```
Object = { value = 0 }
newObj = {}
mt = {}

function Object:add( num )
        self.value = self.value + num
        return self.value
end

function Object:sub( num )
        self.value = self.value - num
        return self.value
end

mt.__index = Object

setmetatable( newObj, mt )

newObj:add( 20 )
print ( newObj.value )
-- outputs 20
newObj:sub( 15 )
print ( newObj.value )
-- outputs 5
```

Here, the Object table provides the definition for several functions, while the newObj object instance implements the functions. The implementation was achieved by simply assigning the 'class' definition as the value of the __index metamethod.

Assigning Values with the __newindex Metamethod

So, we've seen how Lua provides a metamethod for supplying functionality when functions and properties are accessed, but what about when properties are assigned a value? Well, unfortunately, Lua doesn't provide a metamethod when overwriting data in a property, but it does provide a metamethod for new properties created the first time by assignment; the __newindex metamethod.

__newindex is useful when you want to control what properties can be added to an object. As tables are so flexible, it is easily possible for objects to be extended in all manner of ways that might not be desirable to the purpose of the object. Therefore, when an assignment is captured, it can be discarded if it doesn't meet with the object criteria. For example:

```
Object = { value = 0 }
mt = {}

mt.__newindex = function( tbl, key, val )
	if key ~= "soleProperty" then
		return
	else
		rawset( tbl, key, val )
	end
end

setmetatable( Object, mt )

Object.newProperty = 55
print ( Object.newProperty )
-- outputs nil
Object.soleProperty = 55
print ( Object.soleProperty )
-- outputs 55
```

Another use of the __newindex metamethod is to facilitate a non-existent property, much like with the __index metamethod. However, here, when a value is assigned to a pseudo property, it could potentially update actual properties within the object:

```
Object = { value = 0 }
mt = {}

mt.__newindex = function( tbl, key, val )
	if key == "updateValue" then
		rawset( tbl, "a", val + 5 )
		rawset( tbl, "b", val - 5 )
	end
end

setmetatable( Object, mt )

Object.updateValue = 55
print ( Object.a )
-- outputs 60
print ( Object.b )
-- outputs 50
```

In this example, the value of property a will always be five more than the set numeric value of updateValue, while the property b will always be five less. Note, also, that querying the value of updateValue will always return Nil, as it isn't a real property.

Using rawset and rawget

You may have noticed in the examples above the functions rawset and rawget. These are important functions used when dealing with the __index and __newindex metamethods. The purpose of the functions are to access properties of objects while bypassing any set __index

or __newindex metamethods, so as to remove any chance of entering a recursive loop. For instance, take a look at the following example:

```
Object = { value = 0 }
mt = {}

mt.__index = function ( tbl, key )
	if key == "validProp" then
		return tbl[key]
	end
end

mt.__newindex = function ( tbl, key, val )
	if key == "validProp" then
		tbl[key] = val
	end
end

setmetatable( Object, mt )

Object.validProp = 55
print ( Object.validProp )
```

This example may look logically sound. However, a very big problem occurs when using the property validProp. That is, when assigning a value to validProp, the __newindex metamethod is invoked, which checks that we're accessing a valid property name. This is fine, and perfectly acceptable, but the metamethod then continues to physically apply the value to the property name, which re-invokes the __newindex metamethod, and causes the process to repeat in an endless loop. The same is true of the __index metamethod. So, by replacing these calls with rawset and rawget, we can avoid the recursion while successfully committing or retrieving the validProp value. So, to rewrite the above example, we would do the following:

```
Object = { value = 0 }
mt = {}

mt.__index = function ( tbl, key )
	if key == "validProp" then
		return rawget( tbl, key )
	end
end

mt.__newindex = function ( tbl, key, val )
	if key == "validProp" then
		rawset( tbl, key, val )
	end
end

setmetatable( Object, mt )
```

```
Object.validProp = 55
print ( Object.validProp )
-- outputs 55
```

Creating a Pseudo-Class

To complete this appendix, for those of you who are more than familiar with dealing with classes and objects in other languages, let's create an actual class now showing how it might be instantiated and mimicking more pure OO languages.

Below is a rewrite of our previous dog class:

```
local directions = { north = "northerly",
                     east = "easterly",
                     south = "southerly",
                     west = "westerly" }
local Dog = { mt = {},
                     direction = directions.north,
                     barkMessage = "I'm hungry, feed me",
                     isWagging = false }

function Dog:new()
        return setmetatable({}, self.mt)
end

function Dog:bark()
        print ( self.barkMessage )
end

function Dog:run( dir )
        self.direction = dir
        print ( "running in a " .. self.direction .. " direction" )
end

function Dog:doWag()
        self.isWagging = true
end

function Dog:stopWag()
        self.isWagging = false
end

Dog.mt.__index = Dog

local myDog = Dog:new()
myDog:bark()
myDog:run( directions.south )
myDog:doWag()
print ( myDog.isWagging )
myDog:stopWag()
print ( myDog.isWagging )
```

As you can see, a big improvement over our previous rendition of this class is that the Dog object can now be instantiated. Thus, we can create as many dog instances as we like, each one being fully independent of the last. We also now have a fully embedded metatable so that the definition of the Dog class is fully self-contained. What's more, we are able to provide initial property values that can be adjusted as necessary to suit a given instance.

Summary

This appendix has been pretty heavy, with some very important concepts covered. In a single appendix, we have covered how to achieve a very robust object oriented approach to programming in Lua that will facilitate much of the projects we'll create in this book. We have also covered:

- Global and local variables.
- Variable scope.
- Closures.
- Garbage collection.
- Variable arguments and value lists.
- Function recursion.
- Tables as arrays.
- Working with arrays.
- Looping over arrays.
- Looping over non-indexed key-value pairs.
- Creating objects.
- Using the self property.
- Using metamethods and metatables.