

# Learning Mobile Application & Game Development with Corona SDK



By Brian G. Burton, Ed.D.

***Learning Mobile Application & Game Development with  
Corona SDK***

*Brian G. Burton, Ed.D.*

# **Learning Mobile Application & Game Development with Corona SDK**

By Brian G. Burton, Ed.D.

Copyright © 2014 Brian G. Burton, Ed.D. All rights reserved.

Printed in the Abilene, Texas, United States of America

Published by Burtons Media Group. See <http://www.BurtonsMediaGroup.com/books> for more information.

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

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 (eTextbook): 978-1-937336-07-3 | 1-937336-07-7

Version 1.1.02 (9/2014)

## Table of Contents

About the Author.....	xxiii
Dedication.....	xxiii
Foreword.....	xxiv
Preface.....	xxv
Who This Book Is For .....	xxv
How This Book Is Organized .....	xxvi
Conventions Used In This Book .....	xxvi
Using Code Examples and Fair Use Laws .....	xxvi
Why didn't I use _____ for _____ .....	xxvii
Appendices.....	xxvii
How to Contact Us .....	xxvii
Why I Chose to Indie-Publish.....	xxvii
Chapter 1 Introduction to Mobile App Development .....	1
Learning Objectives .....	1
Introduction to Mobile Application Development.....	1
First, an assumption.....	1
Mobile Operating Systems .....	1
Google Android.....	2
Apple iOS.....	2
Blackberry .....	2
Windows 8 .....	3
Cross-Platform Development.....	3
One Last Note Before We Get Started.....	3
Developing Mobile Applications .....	3
Software That You Will Need.....	5
Setting Up Your Software .....	6
Setting up Your Hardware.....	6
Test Devices .....	6
Android.....	7
<i>iOS</i> .....	7

Book Examples and Graphics.....	7
Editors.....	7
Our First Project: Hello World.....	9
Project 1.0: Hello World.....	9
Project Setup.....	10
Debugging.....	14
Project 1.1: Hello World (v2.0).....	14
Introducing Objects .....	18
Summary.....	19
Programming Vocabulary:.....	19
Questions:.....	20
Assignments.....	20
Chapter 2 Introduction to Functions.....	21
Learning Objectives .....	21
Name that Object .....	21
Local vs. Global Variables.....	21
How to Code Comments.....	23
Device Boundaries.....	23
Project 2.1 Display Size .....	24
Functions .....	28
Event Listeners.....	29
Project 2.1: Fun with Buttons .....	29
Project 2.2: Fun with Buttons .....	33
Throw in a Little Fancy.....	33
Keeping Track.....	35
How Corona reads the main.lua file .....	36
API Documentation.....	36
Summary.....	37
Programming Vocabulary .....	37
Questions: .....	37

Assignments.....	38
Chapter 3 Animation and Orientation.....	39
Learning Objectives .....	39
Animation .....	39
Boolean Expression .....	39
Boolean Practice.....	40
How to Code Decision Statements .....	40
if-then.....	40
if-then-else .....	41
if-then-elseif.....	41
if-then-elseif-else .....	42
Nested if-then.....	43
Loops .....	43
for-next.....	43
while-do .....	44
repeat-until .....	45
Nested Loops.....	46
Infinite Loops .....	46
Math API.....	47
Project 3.0: Basic Animation .....	47
Alpha.....	50
Now You See It, Now You Don't .....	50
Project 3.1: Alpha Fun.....	50
Orientation change.....	53
Project 3.2: A New Orientation.....	54
Summary .....	59
Programming Vocabulary .....	59
Questions .....	59
Assignments.....	59
Boolean Practice Answers.....	60

Chapter 4 Working with Data.....	61
Learning Objectives .....	61
TextField and TextBox.....	61
The String API .....	62
The Math API.....	63
Project 4.0: What's Your Age?.....	65
Publishing to Device .....	70
Publishing to an Apple iOS Device .....	70
iOS Simulator Build.....	72
Apple iOS Device Build .....	72
Android OS Device Build.....	75
Summary .....	77
Programming Vocabulary .....	78
Questions .....	78
Assignments.....	78
Chapter 5 Working with Graphics.....	79
Learning Objectives .....	79
Vector Graphics .....	79
Project 5: Vector Shapes .....	80
Bitmap Graphics.....	84
Icons.....	85
Android: .....	85
Apple: .....	85
Build.Settings and Config.lua.....	86
build.settings .....	86
config.lua.....	87
Dynamic Content Scaling.....	87
Dynamic Image Resolution .....	88
Scaling.....	88
Masking .....	89

Project 5.1: Masks.....	90
Sprite Sheets.....	91
Project 5.2: Sprites .....	93
Project 5.3 Sprite Animation .....	95
Sprite Control Methods .....	96
Sprite Properties.....	97
Sprite Event Listeners.....	97
Other Uses of Image/Sprite Sheets.....	97
Summary .....	97
Questions .....	98
Assignments.....	98
Chapter 6: Creating the User Experience.....	99
Learning Objectives .....	99
The User Experience.....	99
Hiding the Status Bar.....	100
Custom Fonts.....	100
Project 6.0 Custom Fonts.....	101
Groups.....	104
Project 6.1: Group Movement.....	105
Modules and Packages.....	105
Composer.....	105
Project 6.2 A Simple Story.....	106
Project 6.3: Creating a Splash Screen.....	111
Summary .....	114
Questions .....	114
Assignments.....	115
Chapter 7 Working with Media .....	116
Learning Objectives .....	116
Audio .....	116
Sound File Types.....	116

Timing Is Everything.....	117
Streams and Sounds .....	117
Basic Audio Controls .....	118
Duration Audio Controls.....	119
Volume Controls .....	119
Audio Channels.....	120
Project 7.0: Beat-box .....	121
config.lua file .....	122
build.settings file.....	122
Where did I put that file? .....	125
Movies .....	126
Camera.....	126
Project 7.1 X-Ray Camera.....	126
Recording Audio.....	130
Summary .....	131
Assignments.....	131
Chapter 8: A Little Phun with Physics.....	133
Learning Objectives .....	133
Turn on Physics .....	133
Scaling.....	133
Bodies.....	134
Body Types.....	134
Density, Friction, and Bounce .....	135
Body Shapes.....	135
Body Properties .....	136
Body Methods .....	137
Gravity.....	137
Ground and Boundaries .....	138
Project 8.0: Playing with Gravity .....	138
Collision Detection .....	141

Sensors.....	142
Joints.....	142
Pivot Joint.....	142
Distance Joint.....	143
Piston Joint.....	143
Friction Joint.....	144
Weld Joint.....	144
Wheel Joint.....	144
Pulley Joint.....	144
Touch Joint.....	145
Common Methods and Properties for Joints.....	145
Project 8.1 Sample Physics Projects.....	146
8.1A: Touch Joint.....	146
8.1B: Pulley Joint.....	148
Project 8.2: Wrecking Ball.....	151
Trouble Shooting Physics.....	153
Summary.....	153
Assignments.....	153
Chapter 9 Mobile Game Design.....	155
Learning Objectives .....	155
Timers .....	155
More on Touch/Multi-touch.....	156
setFocus.....	156
'enterFrame' Animation.....	156
Project 9.0: enterFrame Animation .....	157
Game Development.....	158
Design Inspiration.....	159
Dragging Objects .....	161
Collision Detection .....	165
Take Your Best Shot.....	166

Reducing Overhead.....	167
Game Loop.....	167
Summary.....	168
Assignments.....	168
Chapter 10: Tables and Arrays .....	170
Learning Objectives .....	170
Tables vs. Tables vs. Arrays.....	170
Introducing Arrays.....	170
Table API.....	172
Concatenation:.....	172
Copy:.....	172
indexOf:.....	173
insert:.....	173
Maxn:.....	173
Remove:.....	174
Sort:.....	174
Flexibility of Lua Array Tables .....	175
The 4 <sup>th</sup> dimension and beyond.....	176
Project 10.1 Multi-dimensional Array.....	176
Project 10.2 Conway's Game of Life .....	177
Summary.....	181
Assignments.....	182
Chapter 11: Going Native - Working with Widgets.....	183
Learning Objectives .....	183
Mock-ups and pre-design tools .....	183
Widgets.....	184
Making Your Widgets Look Good.....	185
widget.newButton.....	186
Project 10.0 widget.newButton Example.....	188
widget.newPickerWheel.....	189

Project 11.1 widget.newPickerWheel Example.....	191
widget.newProgressView.....	192
widget.newSlider .....	193
Project 11.2 Widget Slider & Progress View Example .....	194
widget.newScrollView .....	196
widget.newSegmentedControl.....	197
widget.newSpinner.....	198
widget.newStepper .....	199
widget.newSwitch .....	200
widget.newTableView .....	201
widget.newTableView Example.....	203
widget.newTabBar .....	205
Project 11.4 Widget Tab Bar Example.....	207
Removing Widgets .....	218
Summary .....	218
Assignments.....	218
Chapter 12: System Events & Tools.....	220
Learning Objectives .....	220
System Events .....	220
Accelerometer.....	221
Project 12.0 Accelerometer .....	222
Gyroscope .....	225
Project 12.1 Gyroscope.....	226
Alerts .....	230
GPS.....	231
Project 12.2 GPS .....	231
Maps.....	235
Map Object.....	236
Map Address.....	236
Map Location.....	237

Project 12.3 Maps .....	237
Summary .....	238
Assignments.....	238
Chapter 13: File Input/Output.....	239
Learning Objectives .....	239
File I/O Storage Considerations.....	239
Reading Data.....	240
Implicit vs. Explicit Files .....	240
Implicit Read .....	241
Explicit Read.....	241
Writing Data .....	241
Implicit.....	241
Explicit.....	242
Project 13.0 Reading & Writing to a File .....	242
Project 13.1 Appending & Reading from a File .....	243
JSON .....	244
Project 13.2 File I/O with JSON .....	245
Summary .....	247
Assignments.....	247
Chapter 14 Working with Databases.....	248
Learning Objectives .....	248
Database Defined.....	248
Database Software.....	248
How a Database is Structured.....	249
How to create a Database.....	250
Working with a Database .....	253
LuaSQLite Commands.....	254
Project 14.0: Reading a SQLite Database.....	254
Project 14.1 Writing to a SQLite Database.....	258
Summary .....	269

Assignments.....	269
Chapter 15 Network Communications .....	270
Learning Objectives .....	270
Network Status.....	270
Asynchronous Network Requests.....	271
HTTP.....	271
Project 15.0: Picture Download – Via Network Library .....	272
Socket.....	274
Project 15.1: Picture Download – Via Socket Library.....	274
Web Popup .....	276
Web Popup Example .....	277
Webviews .....	277
Web Services .....	278
Connecting to Proprietary Networks.....	278
Facebook.....	278
Facebook Example .....	279
Advertising Networks .....	280
Summary .....	281
Assignments.....	281
Chapter 16: Head in the Cloud .....	282
Learning Objectives .....	282
Cloud Computing .....	282
Game Services .....	284
Apple Game Center.....	285
Google Play Game Services .....	286
Pubnub.....	287
Project 16.0 Multi-User App.....	287
Corona Cloud .....	291
Summary .....	291
Questions: .....	292

Assignments: .....	292
Chapter 17 Web-based App Development.....	293
Learning Objectives .....	293
When to Build a Web-based App.....	293
Considerations in Web-based App Development .....	294
You say Metrics, I say Analytics .....	295
Project 17: Web-Hybrid app.....	296
Summary .....	304
Questions .....	304
Assignments.....	305
Chapter 18: Advanced Graphics (and a game) .....	306
Learning Objectives .....	306
Graphics 2.0 .....	306
Paint.....	306
Fills - Filters, Composite, and Generators .....	307
Fills.....	307
Filters .....	307
Project 18.1 Filtering with chromaKey .....	310
Composite Effects .....	311
Generators.....	314
Project 18.2 Marching Ants.....	315
Containers .....	316
Project 18.3 Containers.....	317
Liquid Fun.....	317
Project 18.4 Liquid Fun .....	317
A Matter of Perspective .....	320
Project 18.5 Side Scroller: Parallax Scrolling.....	320
Parallax.....	327
Jumping to Conclusions.....	329
Fini (maybe)!.....	329

Summary .....	330
Questions .....	330
Assignments.....	330
Chapter 19 Native Application Development .....	331
Learning Objectives .....	331
Why Android? .....	331
Android Studio.....	331
Project 19.0: Native Android-Hello World.....	332
Summary .....	343
Questions .....	343
Assignments.....	343
Chapter 20 Next Steps and 3 <sup>rd</sup> Party Resources .....	345
Where do we go from here? .....	345
3 <sup>rd</sup> Party Resources.....	345
IDE/Editors .....	345
Lua Glider IDE(Mac/Win) by M.Y. Developers.....	345
Outlaw.....	345
Sublime Text 2 .....	346
TextWrangler .....	346
ZeroBane Debugger (Mac/Win/Linux) by ZeroBane Studio.....	346
Sample Code & Templates.....	346
Glitch Games .....	346
Gymbyl.....	347
LearningCorona.....	347
Outlaw Game Tools .....	347
Roaming Gamer.....	347
Graphics/Physics/Level Editors .....	347
Amino (formerly Spriteloq).....	347
Lime .....	347
Physics Editor by code'n'web – Andreas Löw .....	348

Spine by Esoteric Software .....	348
SpriteHelper & LevelHelper .....	348
Texture Packer by code'n'web – Andreas Löw .....	348
Graphics.....	349
GIMP .....	349
Inkscape .....	349
Blender .....	349
Audio .....	349
Acid Music Studio .....	349
Bfxr.....	350
GarageBand.....	350
Ocenaudio.....	350
App Testing .....	350
HockeyKit .....	350
TestFlight.....	350
Mockup/Design Tools.....	351
BluePrint.....	351
iPhoneMockup .....	351
Mockflow .....	351
Appendix A: Installation of Corona SDK.....	352
Macintosh .....	352
Windows .....	353
Problems? .....	354
Appendix B: Publishing to Apple - Xcode & Apple Provisioning .....	355
Xcode .....	355
Types of Apple Developers .....	355
Apple Provisioning Profiles and Certificates .....	355
Icons.....	356
Corona SDK & the App Store .....	356
Appendix C: Publishing to Android - Installation and Configuring Keystores .....	357

Android Studio Installation .....	357
Icons.....	357
Keystores .....	357
Stores.....	357
Appendix D: The Lua Language .....	358
Lua .....	358
An Introduction.....	358
What is Lua?.....	358
Lua in Practice.....	359
Types and Variables.....	360
Type Declarations.....	360
Nil.....	361
Booleans.....	361
Numeric Values .....	361
Numeric Operators .....	362
Dividing by Zero.....	363
Strings .....	363
Quoting Strings.....	363
Escaping Characters .....	364
Concatenating Strings.....	365
Comparing Values.....	365
Boolean Operators .....	366
The and Operator .....	366
The or Operator.....	367
The not Operator .....	367
Stacking Boolean Operators.....	367
Lua Data Functions .....	368
String Functions.....	368
Finding the Length of a String .....	368
Global Substitution .....	369

Finding a Pattern in a String.....	369
Matching a Pattern in a String.....	370
Obtaining a Characters Byte Value .....	370
Getting a String Value from Bytes .....	371
Changing the Case of Characters .....	371
Retrieving a Segment of a String .....	371
Math Functions.....	372
Function.....	372
Parameters.....	372
Returns .....	372
A Note About Code Blocks in Lua .....	373
Conditional Statements .....	374
The if Statement.....	374
Using else .....	374
Nesting if Statements .....	375
Loops .....	376
The for Loop .....	376
The while Loop.....	377
The repeat Loop.....	378
Using break .....	378
Custom Functions.....	379
Defining a Function.....	380
Returning Values from a Function.....	380
Returning Nothing.....	382
Returning Multiple Values .....	382
Multiple Assignment in Variable Definition .....	382
Multiple Assignment from Function Return Values.....	383
Multiple Return Values as Function Parameters.....	383
Value Lists .....	384
Summary .....	384

Appendix E: Advanced Lua Language .....	386
Lua .....	386
Advanced Topics .....	386
Understanding Variables .....	386
Global and Local Variables .....	386
Understanding Scope .....	388
Functions and Variable Scope .....	389
Closures .....	390
Garbage Collection .....	391
Functions with Variable Arguments .....	392
The VarArg Operator .....	393
Select .....	394
Recursion .....	394
The Table Type .....	396
Associativity .....	396
Tables as Arrays .....	398
Array Indices .....	398
Creating Arrays .....	398
Arrays are Tables Too! .....	399
Unpacking Arrays .....	400
Finding the Length of an Array .....	401
Looping Over Arrays with ipairs .....	401
Adding Values to Arrays .....	402
Removing Values from Arrays .....	402
Converting Arrays to Strings .....	403
Sorting Arrays .....	404
Finding the Largest Index .....	404
More on Tables .....	405
Iterating Through Table Keys .....	406
The next Function .....	406

The pairs Function.....	406
Object Oriented Programming in Lua.....	407
Creating an Object.....	407
Designing Objects .....	407
The self Property .....	409
Metamethods.....	410
Understanding Metamethods .....	410
Registering Metamethods with setmetatable.....	411
Operator Metamethods .....	411
Operator .....	411
Metamethod Signature .....	411
Description.....	411
Accessing Values with the __index Metamethod .....	413
Assigning Values with the __newindex Metamethod .....	414
Using rawset and rawget.....	416
Creating a Pseudo-Class .....	417
Summary .....	418
Appendix F: The Path to Certification .....	419
What is a Corona Certified Developer? .....	419
Why become certified? .....	419
CCD Exam Structure .....	419
The Corona Certified Developer Examination & Requirements.....	421
Section I: The Basics.....	421
Section II:Media .....	422
Section III: Events .....	422
Section IV: Data Storage .....	423
Section V: Native.....	423
Section VI: User Interface.....	423
Section VII: Distribution .....	424



## About the Author

Brian Gene Burton, Ed.D. is a teacher, author, and game developer. Beside writing "Beginning Mobile App Development with Corona" and contributing to several academic books on serious games and learning in virtual worlds, Dr. Burton has created game development degrees at two universities and enjoys researching and playing virtual environments. Dr. Burton 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. Thank you for keeping me focused and not running off on rabbit trails!

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

Special thanks to Loren Dalbert for advice and suggestions on improvements to the early release.

A big thank you to Michael Kelly for providing great advice on how to make several of the chapters more understandable and a better learning experience.

All other graphics (unless specified) and cover designed by Brandon Burton (<http://www.geeklyentertainment.com> ).

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

# Foreword

Welcome to Corona!

Whether you are just beginning or are an experienced programmer, Corona SDK is a fantastic way to develop rich interactive mobile apps. I designed Corona with the principles of play and experimentation. In that way, you'll be able to iterate and build something quickly. For example, in just a few lines of code, you can get objects bouncing off each other.

Today, Corona SDK is the leading mobile development platform for building #1 cross-platform games, apps and eBooks. As a result there's a thriving and supportive developer community that you can join. They come from all backgrounds and experience levels, from indies to game studios, from teenagers to octogenarians, from publishers to agencies.

In this book, you will see that Corona gives you a simple and powerful platform so you can take your idea and build great apps. If you have used other technologies, I think you'll be surprised at how quickly you'll see something interesting on the screen.

Okay, it's time to get started. With Corona's help, I hope you have fun bringing your ideas to life.

Happy coding!

Walter Luh  
Creator of CoronaSDK

# Preface

## Welcome

Welcome to Learning Mobile Application Development! This book is the result of years of developing and teaching mobile application development. When I began writing *Beginning Mobile App Development with Corona* in 2011, it was with a specific audience in mind; my current and future students who are experienced programmers. But before I had even finished that textbook, I had begun hearing from another vocal group, (those who did not have any programming experience but had fantastic ideas for apps that they wanted to develop). Agreeing with Corona Labs founder Walter Luh that everyone should be able to develop mobile apps, I began working on tutorials and lessons for those in this underserved audience.

I selected the Corona SDK to use as the method for teaching programming and mobile app development for several good reasons: Corona by Corona Labs (<http://www.CoronaLabs.com>) was developed from the beginning around the concept that anyone can make mobile apps. Using the Lua scripting language, Corona is easy to learn yet powerful enough to allow you to create great, powerful apps and fast, responsive games. Finally, the international community that has formed around Corona is one of the best, developer friendly environments that I have ever experienced in my 25+ years of programming. Corona User Groups and meet-ups are happening all over the world thanks to a devoted network of ambassadors. If there isn't a Corona group in your community, contact Corona Labs and ask about starting one! I hope that you enjoy learning to develop your own mobile apps!

Best wishes and looking forward to seeing your app in the stores,

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. As I wrote this book, it was with the expectation that this is your first time programming or you are not an experienced programmer. If you are an experienced programmer and would like to learn Corona, I would recommend my first textbook, *Beginning Mobile App Development with Corona*.

## How This Book Is Organized

While writing this book, I have kept the traditional 16-week U.S. college semester in mind, assuming one chapter per week. I have included additional chapters to meet specific state requirements at the high school level. These final chapters should allow faculty to create a course that fits their specific needs or allow high school classes to make use of the textbook for the entire year. 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 28 chapters. After reorganizing content and continuing to develop, we are now down to 19 chapters with an additional chapter on great resources and a couple of appendices for the specific installation and app publishing requirements for Apple or Android. Chapters 17 and 18 are to meet specific Texas state requirements for teaching mobile application development at the high school level. Chapters 17 and 18 were developed so that they can be covered at any time after Chapter 7.

## 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 and Fair Use Laws

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 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 apps appear that exactly reproduce the examples from this book, I will not be a happy camper and will contact the app store that the offending app is a violation of copyright. 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](mailto: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 high school and college students, or people just getting started in app development, I tried not to 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/forum>.

## **Appendices**

Appendix A discusses how to download and install Corona SDK. Appendix B covers configuring xCode and setting up provisioning profiles with Apple. In Appendix C, I cover installing Android Studio and configuring Keystores for provisioning your Android based apps. Appendices D and E were provided by Corona Labs (with minor copy editing provided by Burtons Media Group) and covers the Lua scripting language in greater detail. The final appendix F, is a review for the Corona Certification examination which was written by the author.

## **How to Contact Us**

Please address any comments or questions to the books website:

<http://www.BurtonsMediaGroup.com/books> or email

[DrBurton@BurtonsMediaGroup.com](mailto:DrBurton@BurtonsMediaGroup.com). You will find discussion forums for this and other books at <http://www.BurtonsMediaGroup.com/forum>

## **Why I Chose to Indie-Publish**

The decision to Indie-publish was reached after a great deal of consideration. While there were numerous publishers interested (both academic and technical), I decided to release 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 such pricing places an undue burden upon students and schools. While publishers have cut the price slightly with the advent of eBooks and eTextbooks, it hasn't

been enough in my opinion. By indie-publishing, I am only at the mercy of Apple, Amazon, Google, and Kobo.

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 as an eTextbook, which allows me to update and provide it to you, the reader, more rapidly. I will provide the updates between editions to the eBook to everyone who purchases the eTextbook through my website: <http://www.BurtonsMediaGroup.com/books/book-update/>

However, 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 textbook and encourage me to continue to keep it updated. 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 indie-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. To make things more challenging, 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 via email or through the book's forum site: <http://www.burtonsmediagroup.com/forum> and I will make sure that it is fixed in the next update.



# Chapter 1 Introduction to Mobile App Development

## Learning Objectives

In chapter 1 we will learn:

- The different mobile operating systems
- The life-cycle of a mobile app project
- Software needed to make a mobile app
- How to make your first app
- Troubleshooting basics
- About Objects, Methods, and Properties

## Introduction to Mobile Application Development

You have been working on your killer mobile app idea for days. It is completely original; no one has done anything like what you have planned before! Just one problem... How do you get your idea on the tablet or smart phone?

Don't worry! You are in the right place! This book was specifically written for you! In the following pages we will walk through all the decisions and processes that you will need to address to develop and sell your app.

To begin, we will examine the various options you have for developing your app.

### First, an assumption...

This textbook is designed for the person who has no or very limited previous programming experience. If you are an experienced programmer, I would recommend that you use one of my other books such as "Beginning Mobile App Development with Corona," a textbook that makes the assumption that you understand the fundamentals of programming and are comfortable creating your own loops, decision statements, and functions. If that sounds like a foreign language to you, then you are in the right place!

### Mobile Operating Systems

The smart-phone and tablet world are divided by the operating system (OS) that runs on the device. An operating system handles all of the directions from the apps that are running and what the user is tapping on as well as connecting to the Internet, handling text messages and phone calls. It is a busy system! As we begin to develop applications for smart phones and tablets, it is important that we keep in mind the devices that our apps will be running on. Below I have listed the four most popular Operating Systems for mobile devices:

## **Google Android**

The Android OS was developed by Google. The first beta was released in 2007 and it has been regularly updated every few months. At the time of this writing KitKat (v.4.4) is the current stable release.

Native development for Android devices is done with Google's APK (Android Programming Kit). But many tools (including Corona, which we will be using) also allow you to build android applications.

The Android OS is available on smart phones and tablets. It is the foundation of Amazon's Kindle Fire and Barnes & Noble's Nook tablets and enjoys a devoted following. Android Apps can be sold on Amazon, Barnes & Noble, and Google Play. Watch for Android to begin showing up on gaming consoles such as the Ouya.

## **Apple iOS**

Apple's iOS (previously known as iPhone OS) was first released in 2007. Apple iOS can only run on Apple hardware according to the licensing agreement that you sign when you download the software. At the time of writing v.6 of iOS is now available and has been widely adopted by the Apple community.

Apple iOS runs on the iPhone, iPod Touch, and iPad. Apple iOS apps can only be sold through the iTunes store.

Apple iOS native app development is done with xCode. However, like Android, there are many tools available (including Corona, which will be the primary tool that we use in this book) that allow you to build for multiple operating systems/devices at the same time.

## **Blackberry**

At one time, Blackberry smartphones by Research In Motion were the smartphone to own. They controlled a significant market share prior to the advent of the iPhone and Android smartphones. It is difficult to say the impact that Blackberry will have in the future; they are releasing a new SDK called Blackberry 10 and it is supposed to allow Android apps to run on the platform.

At this point in time, only a few multi-platform app development tools do support Blackberry.

## **Windows 8**

Microsoft has dabbled in the smartphone and tablet arena for years (one of the first tablets used Microsoft). However, it has not found much traction. Microsoft hopes to change all of that with Windows 8 and the release of Surface tablet (which Microsoft refers to as a PC). This is a new Surface tablet. Previously Microsoft called their interactive tabletop Surface. Native app development for Windows 8 devices is done in C# using Visual Studio.

## **Cross-Platform Development**

Perhaps I am just lazy, but I don't like to do things twice. When I have created a great app (or even a not-so-great app) for the iPhone or iPad, I don't want to spend weeks completely re-writing all of the programming code just so that it can be deployed to a different set of devices. When I first got started in app development, this is exactly what you had to do. Fortunately, there are now many tools that allow the developer to create apps for more than one operating system.

We will be using the Corona SDK by Corona Labs (<http://www.CoronaLabs.com>) for the majority of our development. Corona Labs was created in 2008 as a venture-backed company in Palo Alto, California. Before Corona, the Corona Labs 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 Corona Labs to be one of the most friendly and helpful businesses that I have had the pleasure of working with. In addition, their online community is unusually friendly and supportive. If you decide to join the Corona community, be sure to continue this great spirit of helpfulness!

At the time of this writing, Corona supports iOS & Android app development, with additional operating system support planned for the near future.

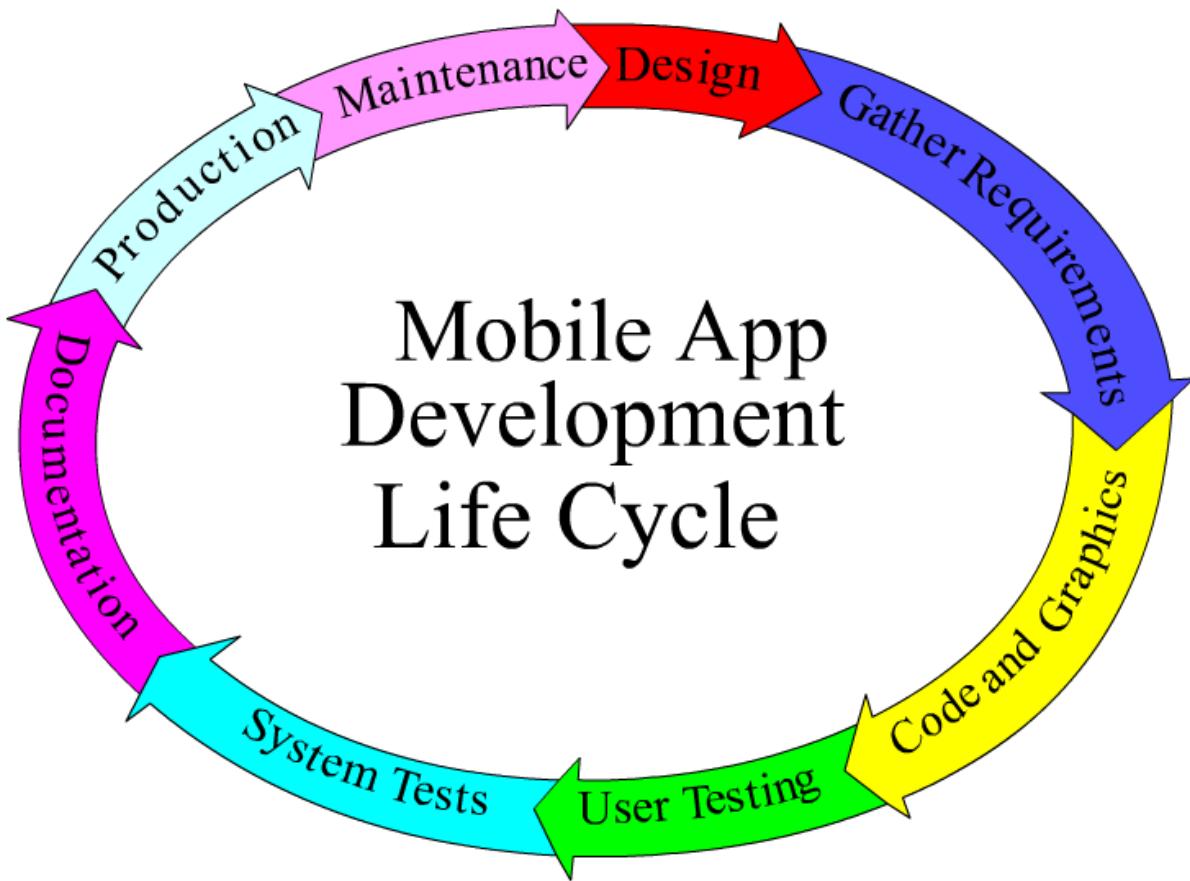
## **One Last Note Before We Get Started**

Corona Labs has recently begun offering the option of becoming a Corona Certified Developer (CCD). As the author of the test and study guide, I want to let you know that if you successfully complete this course, you will have covered everything you need to become a CCD except one; publishing your own app to a store, but we'll cover how to do that so that you can earn your CCD.

## **Developing Mobile Applications**

From concept to store, an app development project goes through eight stages:

- 1) **Design phase** – This is the entry point that many apps never get past. You have an idea for what you want the app to do. The first thing to determine is if the device you hope to place the app on can even do the tasks that you are requiring of it. If it can, is the project feasible? There are many considerations including development cost, software cost, and time; I think you get the picture. If the project appears to be capable of being completed, then you move on to the next stage. Remember to include in this initial design phase how to make money from your app and include any social networking/marketing ideas. The most successful apps plan for marketing and sales from the beginning. One final consideration in the design phase is legal. Be sure to investigate intellectual property and regional/national laws where you hope to sell your app.
- 2) **Gather Requirements** – This phase is about detailing exactly what functionality the app will contain and the design of what the various views will look like (also called storyboarding). This is an essential phase. If you, the developer, are not sure what the app will look like when it is completed, how can you communicate what you are developing to others? Be sure to keep these initial designs and develop a webpage around the development process. It will help with your marketing efforts!
- 3) **Code and Graphic development** – This is where you get started programming and developing the graphics for your project. The best teams are composed of programmers and artists.
- 4) **User Testing** – Too many people skip the testing phase or do not conduct a thorough enough test of their app. Deploy your app to a few test devices and have people in your target audience use the app. Listen to their feedback (remembering that they might be overly kind) and implement their suggestions.
- 5) **System Tests** - Before releasing your app into the wild, run a systems test. Is your app connecting to a remote server or the cloud? Facebook? Twitter? Make sure that all of these features and services are capable of supporting the additional demand your app might place on it.
- 6) **Documentation/Marketing** – Before you can release your work to an app store, you must have a supporting webpage in place with contact emails and screen shots. This is also part of your marketing effort so make sure everything is perfect!
- 7) **Production** – Create your app for release and place it in the stores in which you would like to sell it.
- 8) **Maintenance** – If you haven't noticed, the operating systems for devices are constantly changing with more, newer devices becoming available on almost a daily basis. You should expect to refresh and update your app at least every few months at the very least. On the bright side, in most app stores, releasing a new version could get you a higher ranking in the search engine!



The above list is based upon the traditional software development lifecycle. I have made one adjustment. In the traditional lifecycle, Documentation/Marketing is placed after Production. In app development, that would be a mistake. Documentation and marketing are too important and their not being completed will delay you being able to submit your project to the app stores.

### Software That You Will Need

It's no surprise that you will need the Corona SDK to get started. For learning, I recommend signing up and downloading the Starter version. Just head over to the Corona Labs website: <http://www.CoronaLabs.com/store>. Click on the buy button and register (whether you are purchasing the Pro subscription or using the free Starter version). If you are a student or faculty, you can get a discount on a Pro subscription by going to <http://www.coronalabs.com/store/corona-for-education/>.

## **Setting Up Your Software**

The process varies depending on whether you are installing Corona SDK on a Macintosh or a Windows computer system. A full tutorial for installing the software on both machines is provided in Appendix A. In Appendix A we will also discuss what software is required to publish to devices for different operating systems.

## **Setting up Your Hardware**

Corona isn't too demanding on your development computer. As long as you are running at least OSX 10.7 (Lion) 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 must have a Mac of some type to publish 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:**

<b>Development Hardware</b>	<b>Android OS</b>		<b>Apple iOS</b>	
	Develop	Deploy	Develop	Deploy
Macintosh	X	X	X	X
Windows PC	X	X	X	

## **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 aforementioned 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 deploying to an iOS device, 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 newest 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.

## **Book Examples and Graphics**

If you don't want to create your own graphics or you would like to double check what you have programmed against what I have coded, I have created a repository of code samples, 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/learning-mobile-application-development/>.

## **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) Outlaw, Sublime, 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.

BBEdit does a nice job for a multitude of editing needs. 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>

### **Outlaw (Mac/Win)** by J.A. Whye, free or \$29.99.

Outlaw has a built in editor and is a great tool that I have used for editing and project management. Coupled with its ability to greatly simplify tracking your Corona project, the

cost of Outlaw is well worth it. See Chapter 20 for a coupon code to save on Outlaw.  
<http://outlawgametools.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>

**Lua Glider IDE** (Mac/Win) M.Y. Developers, \$39.99

Glider is a popular integrated development environment (IDE) with some really nice features for troubleshooting and app development. Features include pause and continue, the ability to set break points and stepping through code. A really great tool!  
<http://www.mydevelopersgames.com/Glider/>

**NotePad++** (Win) Open source, \$0

A popular open source language editor for the PC environment. <http://notepad-plus-plus.org/>

**Sublime Text 3** (Mac/Win) free trial, \$70

Sublime is the official editor for Corona SDK. Corona Labs has created a plugin that simplifies many of the processes. You can download Sublime at:  
<http://www.sublimetext.com/> and the Corona Labs plugin (with directions) at:  
<http://coronalabs.com/products/editor/>.

**TextMate** (Mac) by Micromates, €39 (about \$57).

Textmate is very popular in the Corona community with a Corona plugin available on the Corona Labs 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.

## Our First Project: Hello World

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 Corona Labs 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 make an app that will help you learn your way around Corona: a "Hello World" project.

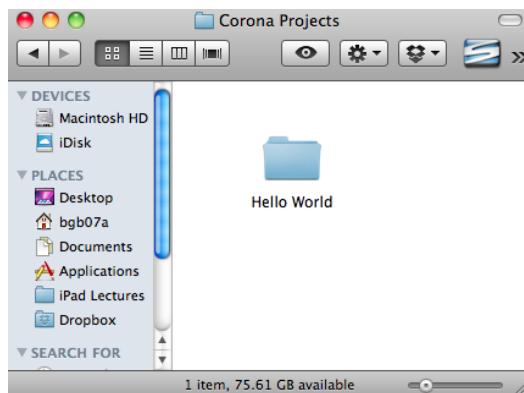
Stop with the rolling of eyes! Before I lose you, let me guarantee that you will get a very valuable resource out of this Hello World project, something that you will use the rest of the time you develop in Corona.

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

## Project Setup

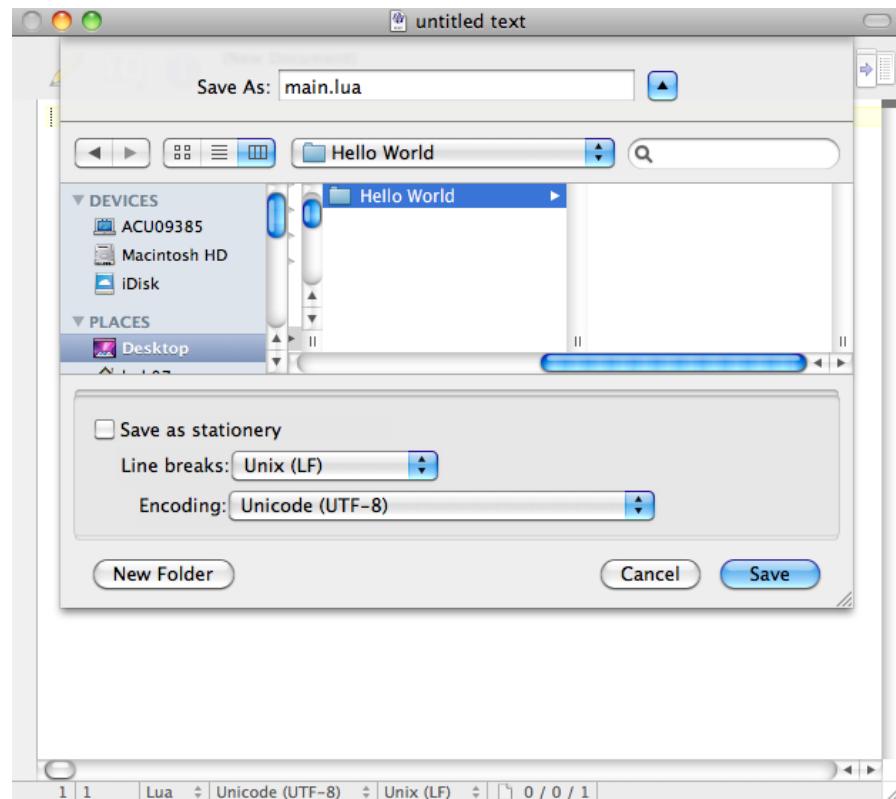
If you follow this process each time you start a new project, it will make your life a lot easier:

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.

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

*Hello World project in the editor*

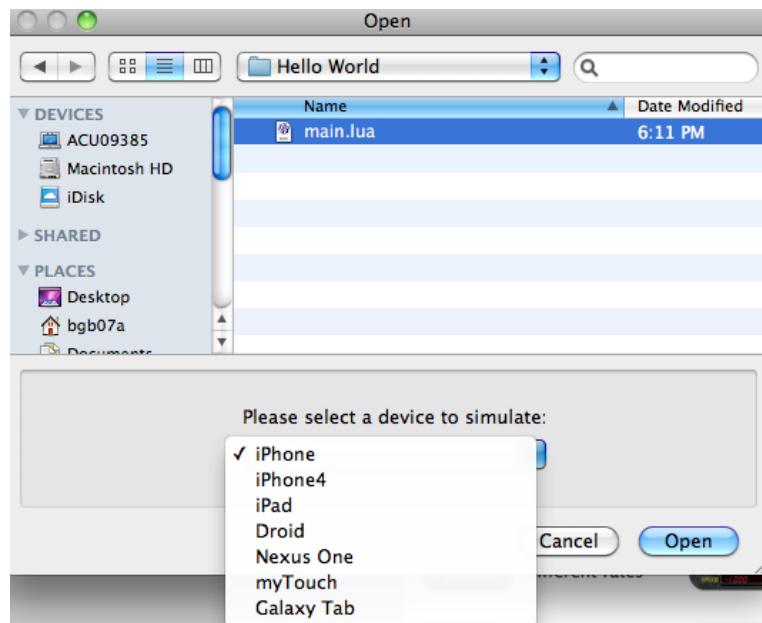
Next, you will need to launch Corona. If you are on a Microsoft Windows system, launch the Corona Simulator. On a Macintosh, launch Corona Terminal.

**An Important Macintosh Note:** Throughout the book I recommend using the Corona Terminal to launch Corona instead of the simulator if you are using an Apple computer. The Corona Terminal can be found through your Finder under Applications > CoronaSDK > Corona Terminal. On Windows, the terminal windows will open automatically when you launch the simulator.



*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.

On the last line of the text in the Terminal you should see your Hello World displayed.

Copyright (C) 2009-2012 Corona Labs Inc.

Version: 2.0.0

Build: 2012.971

The file sandbox for this project is located at the following folder:  
(/Users/bgb07a/Library/Application Support/Corona Simulator/4-

E816C7D29A533635C3F5F861E33B163B)

Hello World

### *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 troubleshooting 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 type.

## Debugging

One of the key methods for debugging is liberal use of the print() command. With the print() command, you can pass a variety of text or variable values to the NSLog (a file that tracks events for Apple devices) or terminal window (depending on your version of Corona SDK). To be a successful app developer, you will quickly come to depend on using print throughout your program.

## Project 1.1: Hello World (v2.0)

Well, that was frustrating! I wanted to make something appear on the screen! Let us make a second attempt at getting something on the simulator screen. Back in your editor (you can use the same main.lua file) type:

```
local textObj = display.newText("Hello World", 100, 100, native.systemFont, 24)
textObj:setFillColor(1, 1, 1)
```

Lua, the language behind Corona, is case sensitive. So newText is a different word than newtext or NewText. Try newtext and look at the error that appears in the Terminal window.

**Note:** Be sure to use " " and not "" in your apps. "" will cause an error!

So what do you make of everything you just typed? Take a moment and savor the possibilities. What will the screen look like when it launches? Why?

Of course we expect “Hello World” to be emblazoned somewhere on the screen, but where? Perhaps those two 100’s have something to do with where it will be placed on the screen? What will it look like? Looks like some sort of default font will be used for the text, and we are setting the color of the text, but what color shall it be?

I’m sure you have a lot of questions on what you typed in. Time to save and launch your app, and then we will look at what happens.

Save the file, and then launch your simulator. You should now see Hello World displayed in the simulator:



*Hello World* on the Droid simulator

Looks like it's an equal distance from the top and the side:  $100 \cdot 100 / 100 = 100$  pixels either way, right. It's definitely a workhorse system font, probably Helvetica, and those three 1's somehow add up to white. Okay, so I'm presuming you just came out of a cave and know nothing about RGB values. Well, just in case you did, let's change a couple of those 1 to 0's and see what happens:

Hello World  
(1, 0, 0)

Hello World  
(0, 1, 0)

Hello World  
(0, 0, 1)

And of course when you add Red, Green, and Blue together in the additive color system, you get White.

Now that final number: 24. Let's change it to a larger number such as 48. What do you think will happen?

What if you want to change the font to say, *Times*. What would you do?

Well, first off, you'd obviously have to get rid of the offending `native.systemFont`, but could you just type in *Times*? Let's try it, change `native.systemFont` to *Times*.

Hmmmmm. Nothing. No error in the terminal window, but no change on the display either.

I will give you a clue: *you can quote me.*

We have to convert `Times` to a string (i.e. place it inside of quotes) for it to be understood. Fonts are stored by their literal names and can only be recognized if you tell Corona that this is a name, not a variable. Anything placed in quotes is referred to as a string (which we will discuss in greater detail later). The only way the computer in our smartphone knows that we are looking for the name of a font is if we place it in quotes. Try changing the font to "Times" and see what happens.

Now for the technical explanation of what we just accomplished:

First we created a local variable called `textObj`. A **variable** is just a place holder for other things in our program. Often times we aren't sure what the value or information will be when we are writing our apps, so it is necessary to use a variable to hold our information. Remember back in math (or if you are from the U.K., maths) when you would use `x` or `y` to solve an equation? Well, a variable is the same thing, except we are going to name our variables much better than `x` or `y`. Good variable naming will make our lives much easier when we get to more complex apps that might have 10 to 20 (or more) variables. 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 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.

We set `textObj` equal to the **object** (which we will discuss in just a minute) that we create by calling `display.newText`. `display.newText` is a command that Corona understands. When Corona sees `display.newText`, it knows that we are going to type something to the screen by telling it what we want to type, where we want it placed (the numbers 100, 100, which are the center X and Y of the text), what font we want to use, and how big the text will be.

The `display.newText` parameters are:

```
display.newText(text, center x or width, center y or height,  
font, text size)
```

or

```
display.newText("Hello World", 100, 100, native.systemFont, 24)
```

In the second command line,

```
textObj:setFillColor(1, 1, 1)
```

we set the color of the textObj that was just changed using the R, G, B color system (each color (red, green, blue) having a value between 0 and 1) to white (which is 1, 1, 1; to get black, we would set it at 0, 0, 0; red is 1, 0, 0; green is 0, 1, 0; blue is 0, 0, 1):

```
textObj:setFillColor(R, G, B)
```

By default, the text object is white, so we didn't really accomplish anything by setting the textObj to white. But 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 mobile 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 “ ”.

## Introducing Objects

You may have noticed the use of the term **object** sprinkled throughout the book thus far. When I use the term 'object' it is to represent 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 you or the people using your app can interact with, including viewing, tapping, dragging, listening to, or just a pretty picture that is on the display.

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 object's properties to make changes; such as when the textObj was created, we set the center X, center Y, font, and size properties as well as the string that would be displayed.

Most objects can have their property changed just by setting it to a new value:

```
textObj.x = 100
```

would move the Hello World that was displayed on the screen to pixel location 100 (or to the right 50 pixels of the original location). Properties always have a period between the object name and the name of the property.

A few valid properties for `display.newText` include:

`object.size` – set the font size of the text

`object.text` – set or change the text

`object.x` – set or change the x location of the object (based upon the center of the text)

`object.y` – set or change the y location of the object (based upon the center of the text)

Objects can also have **methods**. A method is something that changes the current state of an object. Think of a lamp. A lamp can be turned on or off. If we were going to have a method for a lamp, we might call it `setLight` so that we could have it on or off.

To use a method, we put a colon between the object's name and the method we are going to use. In the case of our text, the primary method that we are concerned about right now is `setFillColor`. To change the color of the text we would use the command

```
object:setFillColor(R, G, B)
```

Okay, that is enough for now! If it seems confusing, do not worry about it, it is confusing when you are first getting started! Give yourself a little bit of time to get used to the idea. Remember: 75% of any new skill is learning the vocabulary. If you get used to the idea that an object can be anything in our app and a variable is just the name that we are going to use to refer to that object, you are most of the way there already!

## 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 (okay, maybe not saleable apps, but they are apps)! The first app introduced the critically important `print` command; the second app actually displayed text to the Corona simulator, our original goal. Finally, the concept of a variable and an object in programming was briefly introduced. If the idea of an object and variable doesn't seem natural yet, don't worry, it will make more sense as we learn more material.

## Programming Vocabulary:

Method

Object

Property  
String  
Variable

### Questions:

1. What is a method? Specify its importance and give an example.
2. What is an object? Specify its importance and give an example.
3. What is a property? Specify its importance and give an example.
4. How are objects and properties related to one another?
5. What is a variable? Specify its importance and give an example.
6. What is the programming language that Corona uses?
7. True or false: I can publish for an iOS device using a PC.
8. True or false: Corona allows me to publish to multiple mobile operating systems.
9. List and summarize the eight steps of app development.
10. When developing an app with Corona, what should your first file be named?

### Assignments

1. Try various typos to see the resulting error messages in the terminal window.
  - a. Make a typo in newText. What is the result?
  - b. Make a typo in native.systemFont. What is the result?
  - c. Try setFillColor. What is the result?
2. Change the text object to red in the Hello World (v2) project.
3. Reposition the text to the bottom of the simulator without letters going off the bottom by changing the x and y values of display.newText in the Hello World (v2) project.
4. Place 5 different messages in different places on the screen, each in a different font, size, and color. Note that fonts will depend upon your system. Remember that the font name must be enclosed in quotation marks. Don't worry, if the font is not available, the system will switch to a default font.

# Chapter 2 Introduction to Functions

## Learning Objectives

In chapter 2, we will learn

- about Variables
- the difference between a local and global variable
- how to place a comment in your program
- how to determine the screen size of a device
- how to create a routine that can be used later
- what a listener is and how to use one
- about the API and how to use it

## Name that Object

In our last chapter we learned about objects. Now it is time to name the object. Whenever we assign an object to a variable, we are essentially giving that object a name. Everything that the object is will be stored and referred to through that assigned name. Just as you are called by a name, so too are the objects within our app. When I assign text that will be on the screen to the variable name `textObj`, we have given it a ‘name’. Using this name (in this case `textObj`), we can give it further instructions later in the app.

## Local vs. Global Variables

Most of us have the experience at one point or another where we are given a ‘temporary’ name. Perhaps during school there were two students with the same first name in a class, so they would have their last initial also used: thus two Heathers became HeatherA and HeatherB. Or maybe you were given a nickname in school or in athletics. Usually these different names were short lived or only used in limited situations.

There are a few rules to the naming of variables:

- A variable can be any combination of letters, numbers, or underscores
- A variable must not begin with a number
- A variable cannot contain a space or any symbol except underscore
- Variables are case-sensitive. `myVariable` is not the same as `MyVariable`

Valid	Invalid
Variable1	1Variable
My_Variable	My Variable
_variable	-variable
variable12345	variable12.345

In programming we have two types of variables: local and global. As we progress through the rest of this book (and any other programming language that you ever learn) think of the differences this way: a local variable is a short-lived name given to an object, much like that short-lived nickname in school. A global variable is a long-lasting name and can be used throughout a program, anywhere in a program.

How do you tell them apart? Easy. A local variable will always have the word **local** in front of it the first time it is used in an app. A global variable will never have the word local.

Declaring a local variable:

```
local textObject
local myNewPicture
local backgroundImage
```

Declaring a global variable:

```
textObject
myNewPicture
backgroundImage
```

The preference in programming is to always use local variable whenever possible. Local variables use less memory and will help you avoid naming problems in more complex programs.

One last thing before we move on: while you can place text and images on the screen without using a variable, you won't be able to move, hide, change, or remove them later. It is best to always use a variable name for your objects.

```
display.newText("Can't Touch This", 100, 10, nil, 16)
local myObject = display.newText("Can Touch This", 100, 50, nil, 16)
myObject.y = 100
```

## How to Code Comments

If you have never written a program before, placing comments might seem like a silly waste of time, especially when you are working on a simple program. Let me assure you, comments are as important as any line of code that you write! While good commenting is needed in all software development, it is especially critical in mobile app development. Mobile apps have a very short cycle before they have to be updated. Usually you will be updating a successful app every 6 to 12 months; just enough time to completely forget why you wrote a line in the program to do a specific operation or how you used a special command to fix a bug.

Comments are a gift you give the future you (or the programmer who comes after you to update the software). Taking a few minutes to leave good comments in your program will potentially save you hours of work later. Learn the habit of commenting now. Believe me, you will thank yourself later!

There are two ways to comment in Corona: line comment and block comment.

Line comments convert the remainder of the current line into a comment. By placing two dashes: -- you tell Corona that everything after the dashes is a comment and can be ignored.

A block comment also begins with two dashes and is followed by two brackets: --[[ You end the comment with: --]] usually placed on its own line in the editor. Everything between the brackets will be ignored by the compiler.

Block commenting is a great way to turn on or off a section of program when you are doing testing.

```
-- This is a comment.  
local myObject = display.newText("Your Text Here", 10, 40, nil, 20) -- everything after the 2 dashes is a comment  
  
--[[  
  
This is a block comment. Everything between the double brackets is a comment.  
  
--]]
```

## Device Boundaries

Developing an app for mobile devices is a little different than traditional programming. One of our considerations, especially when we are developing apps for a number of devices is that each one has a different screen size and pixel count. Even if you are just going to publish to Apple, you have four different resolutions that need to be supported. If you add

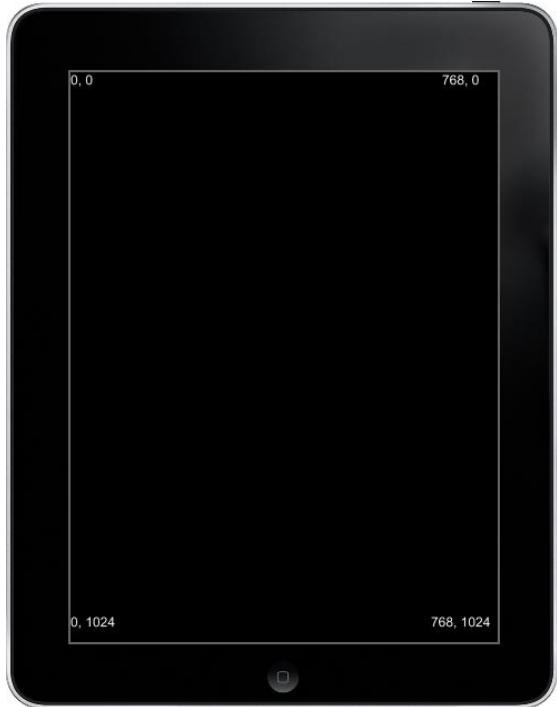
all the different Android based devices, you could drive yourself crazy making all of the adjustments so that each view that is created looks good.

One way around this problem is to use variables to store the device resolution. Corona has built-in commands that will return the display width and height. The command `display.contentWidth` and `display.contentHeight` will return the width and height. These are just two of the commands included in the Corona SDK Application Programming Interface (usually abbreviated as API). All programming languages and development kits include an API so that programmers know what commands are available and how to properly use those commands.

### Project 2.1 Display Size

For our first project in this chapter, let's create a simple program that will give the screen size of our device. Specifically, I would like to know what the x, y location is for each corner of my screen. For this program, I would like it to display at each corner what that screen location is, shown in the examples below:





*The iPhone 4, Kindle Fire, and iPad 2 display resolutions (not to scale)*

To get started, we will need to create a new folder for our app. The folder can be any place on your computer, even a flash drive. But you should always create a new folder for each app. Once you have your folder, create a new main.lua file with your editor.

To get started we first need to hide the status bar. This is done using the `display.setStatusBar` API command. To make the status bar hidden, we need to pass it a **parameter**. A parameter is any information that is given to an API command. In this case, the parameter is `display.HiddenStatusBar`. This makes the command to hide the status bar from view during the execution of your app: `display.setStatusBar( display.HiddenStatusBar )`

### main.lua

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

Now that the status bar is hidden, it is time to find out the height and width of the display. We can store the display's width and height in variables for easy reuse with the `display.contentWidth` and `display.contentHeight` commands.:

```
local myWidth = display.contentWidth  
local myHeight= display.contentHeight
```

We now know (well, our app knows) the maximum width and height of the display. Displaying it is just a matter of using the `display.newText`, right? Remember, the top, left corner will be our starting point, so it has an x of 0 and a y of 0.

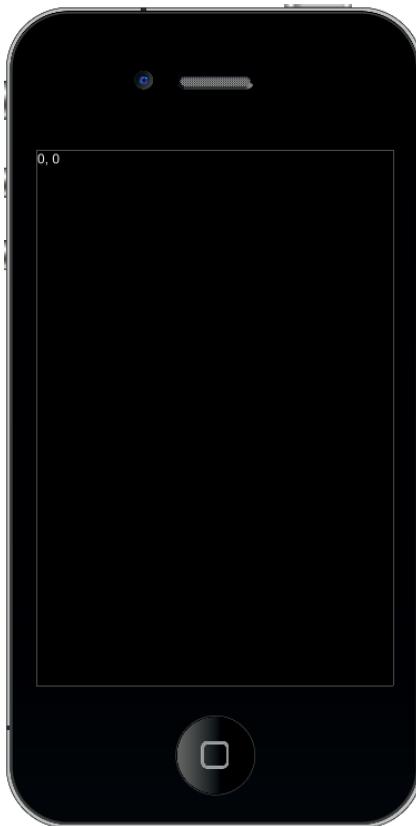
One last thing: instead of using `system.nativeFont` as the fourth parameter of `display.newText`, I have used a short cut: `nil`. `Nil` is nothing, zero, nada. It is a shortcut way of telling Corona that I don't care what system font is used, so Corona will automatically use the system's native font.

```
local topLeft = display.newText("0, 0", 0, 0, nil, 24)
local topRight = display.newText(myWidth.. ", 0", myWidth, 0, nil, 24)
local bottomLeft = display.newText("0, " .. myHeight, 0, myHeight, nil, 24)
local bottomRight = display.newText(myWidth ..", ".. myHeight, myWidth,
myHeight, nil, 24)
```

Did you notice my use of two periods in the `display.newText (myWidth.. ", 0")`? By placing two periods together, we are telling Corona that we want to **concatenate** or take two strings and make them one string. In this case, we are taking `myWidth` and concatenating it with `, 0` which will place the width beside the coma in the string (for example: if the width is 360 pixels, the resulting string would be "360, 0"). We will talk about concatenation more in chapter 4.

Save your app and run it with the Corona Simulator.

Hmmm, the text is only partially showing. That will not work.



iPhone 4 showing only left, top text

Why? As we saw in Chapter 1, `display.newText` positions the center of the text at the values we give. So when we assign `topRight` to `myWidth`, `0`, the `myWidth` positions the text partially off of the screen. Same situation for `myHeight` at the bottom of the screen, it causes the text to be displayed below the bottom. To fix this problem, we need to move the text back on the display by subtracting from the `myWidth` & `myHeight` values. Remember that the measurement is in pixels, not characters. Let's replace the last four lines with the following:

```
local topLeft = display.newText("0, 0", 25, 20, nil, 24)
local topRight = display.newText(myWidth.. " , 0", myWidth-100, 20, nil, 24)
local bottomLeft = display.newText("0, " .. myHeight, 50, myHeight-50, nil,
24)
local bottomRight = display.newText(myWidth .. , .. myHeight, myWidth-120,
myHeight-50, nil, 24)
```

Run the app. You can change the device shown by the Simulator by clicking on the View Menu, then View As. Doesn't that look better? Try changing the values in `left` and `top` parameters of the `display.newText` to see how the text changes location on the screen.

## Functions

A function is a small segment of program that we might need to use at certain times in our program. Say you have created an app that has a button on the screen. In this app, every time the button is tapped, the button is moved to a new location. Now, we don't want this app just randomly moving the button. Nor do we want it continuously moving the button. We only want the button to move after it has been tapped. This can be managed by creating a function.

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.

Always use an original name for your function (and your variables!) that describes what the function accomplishes. It is a common mistake to name a function the same as a variable name that you are using. While you can sometimes get away with it, it is bad programming practice and will create confusion as you begin to create more complex programs. Think of it like this: how confusing is it in a classroom when you have 5 people named Ashley or Jacob?

A function can be implemented in several ways. The most common is:

```
function functionName ()  
    body or list of commands  
end
```

Functions can receive parameters (contained in the parenthesis) and return information to the command that called them using the key word **return** (which we will talk about much later).

Remember that local variables are limited to their current function. If a variable is declared as local within a function, it is only available when that function is in use. Once the function is done, the variable and any information that was stored in it are gone, poof, no longer available. If a variable is declared as a local variable outside of a function, it is limited to the current file (such as main.lua); thus it would be available to all functions within main.lua, but not in other files that might be a part of the application (something we will talk about in a couple of chapters).

## Event Listeners

Event listeners are essential to every app. An event listener is a function that is constantly running in the background on the device. When the event occurs, the event listener tells the appropriate function, which function should handle the event. Think of this like your telephone. It is always ‘listening’ to see if you have an incoming phone call. It only rings when you receive a call. Your phone is like an event listener which then passes control of the phone ringer function when there is a phone call.

Event listeners are usually placed at the end of the program listing since they call functions that must already be declared.

*objectVar:addEventListener(interaction, eventListener)*

## Project 2.1: Fun with Buttons

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. Then I copied the button.png file into a new folder that I named Project 2.1. Once you have a button, and your folder for this project, you will create a new main.lua file in the same folder.

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 creates a local variable called myButton, and then assigns an image to it (the button.png graphic we just created). The command display.newImage will take any image we give it and display it to our screen. I recommend that you always use the PNG file format. PNG is accepted by Apple and Android. Other graphics file formats are not accepted by both.

If you save the main.lua file and run the simulator, you should see the button you created located in the left top corner of the simulator. You could also set the top left corner location of the graphic by adding a setting for the center x and center y of the object as we did with the textObj in chapter 1:

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

We can also set the myButton object location directly by changing the x and y property setting. When you set the x & y values by changing the property, 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:

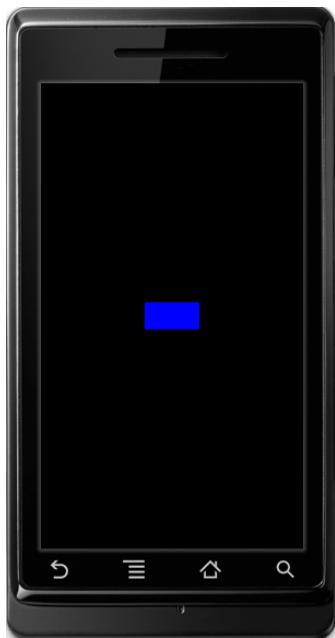
```
myButton.x = 100 -- sets the center 100 pixels from left  
myButton.y = 100 -- sets the center 100 pixels from top
```

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 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, no matter what device it is running on.



*myButton is now in the center of the screen*

Next, we will make something move to a new location on the screen. For this, we will simply place some text on the screen that we can move with our function:

```
local textObj = display.newText("Button Tapped", 100, 50,  
native.systemFont, 24)  
  
textobj:setFillColor(1, 1, 1)
```

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 moveButtonDown( event )  
textObj.y = textObj.y + 50  
  
end
```

This little function will take the old .y property of textObj and add 50 pixels. (i.e., it will move the text “Button Tapped” down the screen 50 pixels every time the button is tapped).

One last line is required before we test our app. Add this to your program as the last line of code:

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

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. We will talk about other types of events at a later time.

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’.

The full program should look like this:

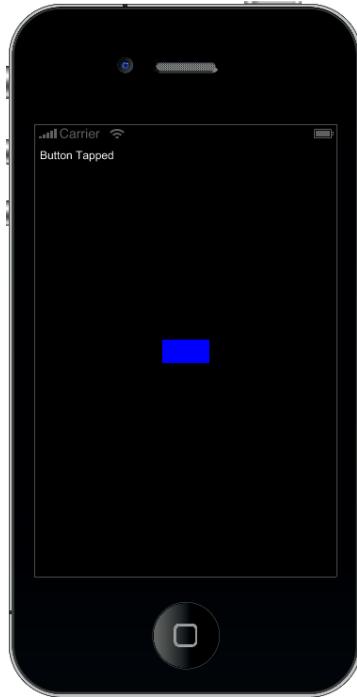
```
local myButton = display.newImage( "button.png" )  
myButton.x = display.contentWidth / 2  
myButton.y = display.contentHeight/2  
  
local textobj = display.newText("Button Tapped", 100, 50,  
native.systemFont, 24)  
textobj:setFillColor(1, 1, 1)
```

```

function moveButtonDown( event )
    textobj.y = textobj.y + 50
end

myButton.addEventListener( "tap", moveButtonDown )

```



Let's adjust the button a little more. To simplify the interface, I want to move the button to 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 that 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 (i.e. 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 a user's finger might be too 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 the right button size for your app. Remember, once you have created a button you like you can reuse it for other projects.

## Project 2.2: Fun with Buttons

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, we can now relocate the text object to a new, random, location.

Your myButton:tap function should now look like:

```
function moveButtonDown( 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.

## Throw in a Little Fancy

Did you notice that sometimes the text object (your "Button Tapped" text) 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 we finally 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 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 troubleshoot 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

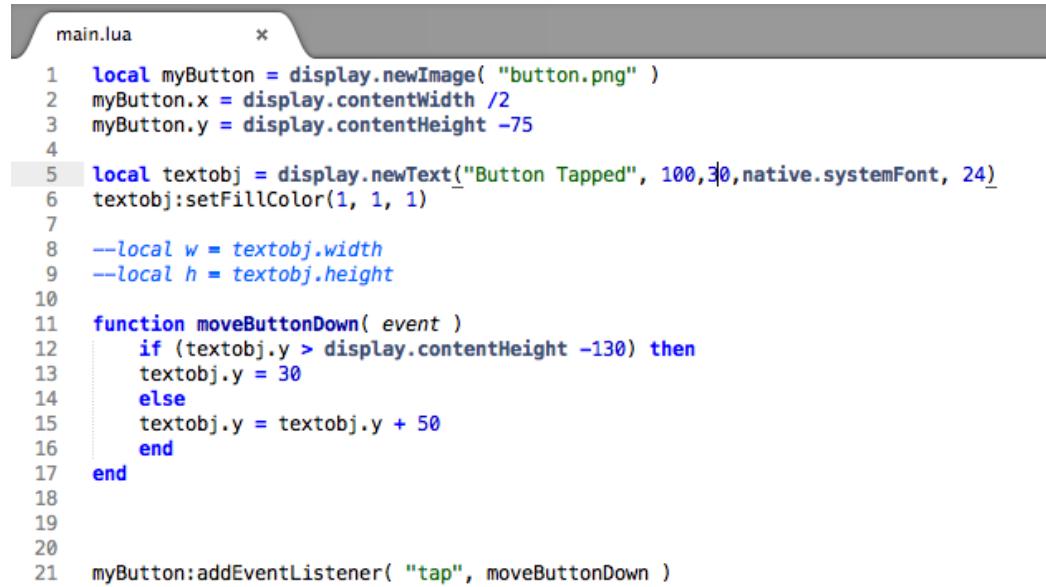
function moveButtonDown( event )
    textObj.x = math.random( w/2, display.contentWidth - (w/2) )
    textObj.y = math.random( h/2, display.contentHeight - (100 +
h/2) )
end
```

**TIP:** Sometimes I have to split the programming code between two lines such as above due to book margins. When you see this happen, the program code should be on one line in your programming editor.

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!



The screenshot shows a code editor window titled "main.lua". The code is written in Lua and defines a button and a text object, then adds a tap event listener to move the text object.

```
1 local myButton = display.newImage( "button.png" )
2 myButton.x = display.contentWidth /2
3 myButton.y = display.contentHeight -75
4
5 local textobj = display.newText("Button Tapped", 100,30,native.systemFont, 24)
6 textobj:setFillColor(1, 1, 1)
7
8 --local w = textobj.width
9 --local h = textobj.height
10
11 function moveButtonDown( event )
12     if (textobj.y > display.contentHeight -130) then
13         textobj.y = 30
14     else
15         textobj.y = textobj.y + 50
16     end
17 end
18
19
20
21 myButton:addEventListener( "tap", moveButtonDown )
```

Now textObj.x property is limited to generating a number between half the width of the text object and the content width minus half the width of the text object. Similarly, the textObj.y property 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 plus 100 pixels (to keep it above the button).

**Tip:** if you have tried to copy and paste from the book into an editor and had strange errors, try retyping the line of code. Sometimes the word processor adds invisible characters.

## Keeping Track

Often times in programming and app development it is necessary to keep track of something such as how many times an operation has been performed (such as how many

times was the button tapped). In the case of moving a text object down the screen, we added a value to the text objects last location. When we want to count something we use a process called incrementing. Basically, it is adding 1 to the previous value. In Lua programming it would look like:

```
local number = 0  
number = number +1
```

In this example we have created a variable called number and initialized it to the value of 0. It is always a good idea to initialize your variables to their starting value. If you initialize a variable but don't assign a value, it will have the current value of nil. If you then try to do a mathematic operation to the variable you will get an error or might have unplanned results.

After we initialize the variable, we can then begin to count operations by increasing the value of the variable. By using `number = number + 1` we are telling Corona that we want to take the original value of number and add 1 to it, storing the new value in number.

## How Corona reads the 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. 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 your functions, and finally, make any needed function calls and add event listeners.

## API Documentation

The API (Application Programming Interface) is the list of all commands that are available in a programming environment. The API is your primary reference tool when you are making apps. Experienced programmers are able to write software in a new program with just the API information. Corona's API is broken into three sections: Libraries, Events, and Types and can be found (and should be referenced regularly due to the rapid changes that occur in the SDK) at: <http://docs.coronalabs.com/api/>.

The [Libraries](#) contain a complete list of built in Corona functions such as `display.newText`. Here you will find the details for the function: Syntax, an overview of what the function

does, parameters that are optional or required, and example code for how the function might be implemented.

The Events column contains all of the different types of events that can be tracked within Corona, as well as their associated properties. Events, like Libraries and Types, are broken into different categories to make finding the proper event easier.

The final column, Types, shows the different data types in Corona, which includes various objects that are produced by different Library functions. Types include properties and methods when appropriate, as well as an overview and sample code.

## 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, functions, and the API.

## Programming Vocabulary

API  
Concatenate  
Function  
Parameter  
Syntax

## Questions:

1. Explain the difference between a local variable and a global variable. How do you tell them apart?
2. List and explain the two different types of comments you can make in Corona.
3. True or false: You should always create a new folder for each app you create.
4. What is a function? Also give an example of a function.
5. What is an event listener, and how is it important to app development?
6. Which image file format should be used since it is accepted by both Android and Apple?
7. What is a button?
8. True or false: The Corona SDK is case sensitive.
9. What is the purpose of Relaunch in Corona simulator?

10. How does Corona process the main.lua file?

## 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 (hint, it is like moving a textObj down 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 moves a text object down the screen by 50 pixels every time a button is tapped.

# Chapter 3 Animation and Orientation

## Learning Objectives

In chapter 3 we are going to begin working with animation and handling when the user changes the orientation of their mobile device. To do this we will learn:

- Methods of animation
- How to program a decision
- How to program a loop
- Fading objects in and out
- Handling orientation change

## Animation

Animation is the process of moving objects on the screen over a period of time. There are a number of ways to create animation effects on mobile devices. In this chapter we are going to look at the most direct method and save the second method for chapter 5. Before we jump into animation, we first need to go over two of the primary building blocks of all programming languages: Decisions and Loops.

## Boolean Expression

Decisions and loops are both based upon the answer from a **Boolean expression**. Based upon the decision generated by the Boolean expression, the program can handle a variety of issues based upon a change in the situation. These decisions are just like they sound: you program into your app the ability to make a ‘decision’ based upon information that is supplied. Unfortunately, computers are not as smart as the movies and television make them out to be. Our mobile device can only make a decision based upon information that is either True or False. This is known as a Boolean expression: there are only two possible answers, but that answer can be displayed as True or False, Yes or No, 0 or 1. We can actually get very complex with Boolean expressions using AND, OR, NOT in a variety of combinations (which we will discuss later), but at the core, all Boolean expressions must be answerable as true or false.

Boolean expressions usually compare one variable to another. To do the comparison, we use the symbols:

`==` - two equal signs to check for equality (two equal signs are used to show that it is a comparison and not assigning a value).

`>` - greater than to check that one variable is greater than another.

< - less than to check that one variable is less than another.  
>= - greater than or equal to checks for one variable being greater or equal to another.  
<= - less than or equal to checks for one variable being less than or equal to the second variable.  
~= - not equal to checks that the variables are not equal (i.e. it is true that they are not equal).

## Boolean Practice

Answer the problems below. All problems evaluate to either True or False.

Assume that A = 1, B = 4, C = 5. (Answers are supplied at the end of the chapter)

1. A == B
2. B > C
3. B < C
4. A <= B
5. C == A
6. A+B == C
7. C-A == B
8. C+A <= B
9. B ~= C
10. B <= C+A

Next, we will look at the basic decision statement syntax and some examples of their use.

## How to Code Decision Statements

### if-then

The most basic decision statement is an if-then. If-then command syntax is:

**if Boolean expression then**  
*one or more statements/commands*  
**end**

The if-then statement is used in all programming languages, though the syntax may vary. In Lua, the decision statement is kept simple, which is to our benefit. An example of an if-then might look like:

```
if (subtotal > 100) then
    discount = 0.1
end
```

In this example, we check to see if the value of the variable subtotal is greater than 100. If it is, then we set a discount rate equal to .1.

**TIP:** Did you notice that the discount = 0.1 line was spaced over from the left column? This indentation is done for readability of our programs. If you space over each time you begin a block of code that is under the control of another program segment, it will make your program MUCH easier to read. This is one of those little tips that will win you big points with your future self! It is usually recommended that you use spaces to move the content over instead of the tab key since a space is a space, but a tab could be different from machine to machine, making things difficult to read. Most programmers place 4 spaces for each level of indentation.

### if-then-else

With the addition of an else situation, we can pass additional commands to the computer:

```
if Boolean expression then  
    one or more statements/commands  
else  
    one or more statements/commands  
end
```

The addition of else tells our program what to do when the Boolean expression is false. An example of an if-then-else would be:

```
if (subtotal > 100) then  
    discount = 0.1  
else  
    discount = 0  
end
```

### if-then-elseif

But what if we need to check for a different situation in case the first situation wasn't met? We have two choices, we can nest an if-then inside another if-then or we can use the if-then-elseif:

```
if Boolean expression then  
    one or more statements/commands  
elseif Boolean expression then
```

*one or more statements/commands*

**end**

With the if-then-elseif, the second Boolean expression is only evaluated in the event that the first Boolean expression was false. Example:

```
if (subtotal > 200) then
    discount = 0.2
elseif (subtotal > 100) then
    discount = 0.1
end
```

In this case, the discount will only be set to .1 if the subtotal is greater than 100 but less than 200. And yes, it is possible to have multiple elseif statements.

### **if-then-elseif-else**

The final decision statement includes the else catch after the elseif. With this statement, we can test for multiple situations and still provide a final set of commands if none of the conditions are met.

**if Boolean expression then**  
*one or more statements/commands*  
**elseif Boolean expression then**  
*one or more statements/commands*  
**else**  
*one or more statements/commands*  
**end**

Example:

```
if (subtotal > 500) then
    discount = 0.25
elseif (subtotal > 200) then
    discount = 0.2
elseif (subtotal > 100) then
    discount = 0.1
else
    discount = 0
end
```

## Nested if-then

Sometimes we run into the situation where it is necessary to place an if-then within another if-then. This is referred to as a nested if-then. Only if the first condition is true will the second if-then be tested:

**if Boolean expression then**

**if Boolean expression then**

*one or more statements/commands*

**end**

**end**

Example:

```
if (subtotal > 500) then
    discount = 0.25 -- 25% discount if sales over $500
    if (customerID == "employee") then
        discount = 0.35 -- 35% discount if sales over $500 and an employee
    end
end
```

There is a lot more that we can do with if-then statements, but we will save it for a later chapter.

## Loops

Loops are also commonly used to control the flow of your program. Often times we will have the need to do a set of commands multiple times or until some condition is met. Loops provide the means to accomplish these repetitive tasks. There are three ways to create a loop: while-do, repeat-until, and for-next. We will examine for-next first, as it is the simplest to understand.

### for-next

The for-next loop is used when we want to execute a block of code a known number of times.

```
for name = start number, end number [, step] do
    block
end
```

*name* - When starting a for-next loop, the first required item is a temporary variable *name* to store the number the loop is on. This can be any variable name, but should not be a variable that you are using elsewhere in your program.

*Start number* - any number or variable that holds a numeric value.

*End number* - any number or variable that holds a numeric value.

*Step* - (Optional) The increment for counting. Example: -1 to count down, 2 to count by evens or odds, 10 to count by 10, etc.

### Examples:

```
for count1 = 1, 5 do -- Count from 1 to 5, printing the result in the terminal
    print (count1) --Output: 1, 2, 3, 4, 5
end
```

```
for count = 0, 10, 2 do -- Count from 0 to 10 by 2's, printing the result in the terminal
    print (count) --Output: 0, 2, 4, 6, 8, 10
end
```

```
for counter = 10, 1, -1 do -- Count down from 10 to 1, printing the result in the terminal
    print (counter) --Output: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
end
```

### while-do

The while-do loop repeats a block of code until a Boolean expression is no longer true. If the condition is false on the first loop of the while-do, then the block of code within the loop will never be run.

**while** Boolean expression **do**

*block*

**end**

**Example:**

```
count = 0  
while (count < 10 ) do  
    print (count)  
    count = count + 1  
end
```

### repeat-until

The repeat-until loop repeats a block of code until a condition becomes true. The repeat-until will always run the block of code at least once since the Boolean expression evaluation comes at the end of the loop.

**repeat**

*block*

**until** Boolean expression

**Example:**

```
count = 0  
repeat  
    print (count)  
    count = count + 1  
until (count > 5)
```

**Tip:** If you need to keep track of how many loops have happened in a while-do or repeat-until, you can add the variable to the block of code that adds 1 each time the block repeats: e.g. counter = counter + 1

## Nested Loops

When you place one type of loop inside another loop, it is called a nested loop. When a nested loop executes, every time the outside (or first) loop does one loop, the inside loop will loop until it completes its required loops.

### Example:

```
for outsideloop = 1, 3 next -- outside loop
    print (outsideloop)
    for insideloop = 4, 1, -1 -- inside loop
        print (insideloop)
    end
end
```

Output: 1, 4, 3, 2, 1, 2, 4, 3, 2, 1, 3, 4, 3, 2, 1

## Infinite Loops

An infinite loop is usually created by accident. It is the situation when the condition that will stop the loop are never met:

```
count = 0
while (count < 10) do
    counter = counter + 1
end
```

As you can see with the example above, count will never increase in value, as the variable being changed is counter. Situations like this will cause your app to lock-up and potentially crash.

To avoid infinite loops, always double check that at some point the condition that will end the loop will occur.

## Math API

We need to be familiar with one more concept before we start our next program: using the math API. Corona provides us with shortcuts so that we are not required to write program code for different mathematical calculations. All of the math API begin with the word 'math' followed by a period, then the function. For example, if you wanted to use Pi in your application, you would only need:

```
local pi = math.pi
```

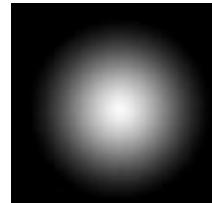
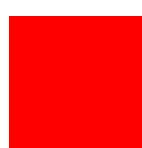
and the constant 3.1415926535898 would be assigned to your variable.

We will discuss the Math API further in the next chapter. For your next project, we are going to be using the math.random function, which returns a random number. The math.random function has three possible uses:

1. If you do not pass it a value, math.random() will return a number between 0 and 1.
2. If you pass it one number, math.random(x) will return a whole number between 1 and x.
3. If you pass two numbers, math.random(x, y) will return a whole number between x and y.

## Project 3.0: Basic Animation

In this first project we are going to use a 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. Both of the graphics should be in the png format which will give us complete compatibility between all of the various devices supported by Corona SDK. (Images and other code samples can be downloaded from <http://www.BurtonsMediaGroup.com/books>).



Create a new folder for this project and copy your png images into the folder.

Open your editor and save a file to your folder as main.lua.

To begin, we will load the graduated white dot and then the square into your app. Place the graduated white dot in the center of your display:

```
--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
```

To place the square at a random location on the screen, we will use the math.random function, passing it the display width and height for the x and y coordinates. Remember that this will generate a number between 1 and the display width and height.

```
-- place the square at a random location on the screen

square.x= math.random(display.contentWidth)
square.y= math.random(display.contentHeight)
```

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. The while loop includes the key word OR. There are three key words that can be included in a Boolean expression: AND, OR, and NOT. For now we will focus on OR and AND.

Including OR in our Boolean expression means that if either condition that we are looking at is true, then the whole expression should be considered true. Using AND means that both Boolean expressions must be true for the expression to be true.

Consider it like this: If someone says “if it rains or snows tomorrow, we will go to the mall.” You know that if it rains or snows, either one, you are going to the mall. However, if someone says “if it rains and snows tomorrow, we will go to the mall.” You know that it must both rain and snow before the shopping excursion will occur. It is the same in our if-then, while-do, and repeat-until programming.

```
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, keep doing 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 would still be far too fast.

Fortunately, there are other options 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 Adobe Flash, then you have probably used tweening. In Corona, we can create similar transitions using the transition.to command. The transition.to API is:

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

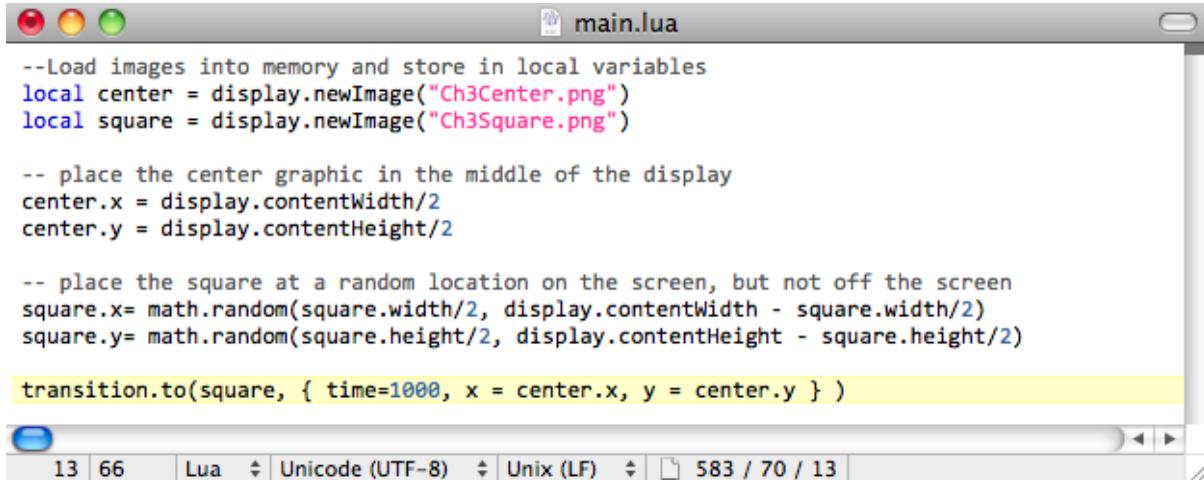
The object is the object we want to move (our square). Within the array of the transition.to, we can pass the parameters we would like to change and how quickly we want that transition to occur.

My new movement code (replacing the old while loop) looks like:

```
transition.to( square, { time=2000, x = center.x, y = center.y } )
```

In this one line of code, I have first passed the variable called square as the object to be transitioned. In the array, I'm passing it a time parameter that is set to 2000 milliseconds

(or 2 seconds), and then the x and y variables with the location that I want to move the square.



```
--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)

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

Want to see a neat feature of Corona SDK? Hold down the CTRL – R (Windows) or Command – R (Macintosh) after you run the app the first time. This will re-launch the app without you having to reload it each time. It will allow you to see the random element for placing the square on the screen.

## Alpha

### 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 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 place 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
```



Now we will need to set up 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.

It works pretty well, except 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.

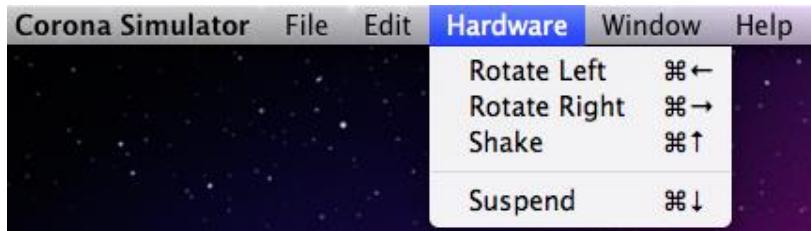
## Orientation change

Device orientation is a very important issue to mobile app stores. They (the users and the reviewers/approval department at Apple, Amazon, Google and any other app store you submit your app) 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, if appropriate.

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 and apps have little problem being approved with having just one or two orientations if it is appropriate to the app.

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.

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:setFillColor(1, 1, 1)
portrait.alpha = 1

landscape:setFillColor(1, 1, 1)
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 )
```

Note that **Orientation change is a system event**. So our event listener is using the keyword “Runtime” to tell Corona to check for the system event orientation. We will discuss other types of system events in later chapters.

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
    end
end
```

```

        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. In practice, the last option is easiest for this project:

```

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:setFillColor(1, 1, 1)
portrait.alpha = 1
landscape:setFillColor(1, 1, 1)
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
```

```
    end
end

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

## Summary

In chapter 3, we learned about Boolean expressions, loops, and decisions, 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.

## Programming Vocabulary

Boolean Expression

Decision Statement

Loop

Syntax

## Questions

1. What app orientations are allowed for a smart phone? For a tablet?
2. What does alpha control?
3. What are three types of loops?
4. How would you make an object look faded?
5. Define Boolean expression.
6. What type of event is an orientation change?
7. Define an infinite loop.
8. Given that you need to sum the numbers 1 through 100, write a loop to complete the operation.
9. List 2 of the operations the operations that can be accomplished with transition.to.
10. What is the difference between using 'else' and 'elseif' when using a if...then statement.

## 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.

## Boolean Practice Answers

Below are the answer to the Boolean Practice

1. False
2. False
3. True
4. True
5. False
6. True
7. True
8. False
9. True
10. True

# Chapter 4 Working with Data

## Learning Objectives

In chapter 4 we will begin working with various types of data:

- How to enter data using textfield and textbox
- Using the String API
- Using the Math API
- How to publish to your test device

Most apps work with data. Sometimes the user enters the data through their simulated keyboard, other times it is through a picker wheel or drop-down lists. In this chapter we are going to focus on working with textfields and textboxes.

## TextField and TextBox

A TextField and TextBox is a part of the native user interface for the Apple iOS and Android devices. What does this mean? That the TextField and TextBox 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.

Okay, and what does THAT mean?

Here is what it comes down to. The TextField and Textbox are handled very differently by Apple and Android. They are ‘native’ tools. Everything else that we do in a mobile app is basically moving graphics around on the screen. But TextFields and TextBoxes receive user input from the built-in keyboard. That means they are different. They are referred to as ‘native’ to the user interface. Since they are native, they have a few different rules that we have to consider when using them.

The first rule that we run into is that we can only test TextFields and TextBoxes (at least at the time of this writing) in three places: in the Corona Simulator on a Mac, in the xCode simulator on a Mac, or on a test device.

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, and then we will tackle each operating system build separately.

Now, let’s look at the difference between a TextField and a TextBox. 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.

The rules are the same for a TextBox. A TextBox is used for multiple lines of text input. It will always appear above (or in front of) all other objects on the screen.

The mistake that I see many students make initially when making a TextField or TextBox is that they do not make it large enough and it cuts off the text that is entered by the user.

The API for creating a TextField is:

```
native.newTextField(left, top, length, height [, handler function])
```

For creating a TextBox, the API is:

```
native.newTextBox(left, top, length, height [, handler function])
```

You also have control as to the type of keyboard that will be called when the user begins to enter data:

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

## The String API

In this next project we will be working a great deal with strings. Strings are created using a matching set of single or double quotes. Anything that is typed into a TextField or TextBox is considered a string, even if it is a number. Fortunately, we have a lot of great tools that make working with strings easy.

The string API includes:

- string.byte() – returns the string ASCII code for the string character specified.
- string.char() – returns the character for the given ASCII code.
- string.find() – returns the start and end locations for a matched string within a string.
- string.format() – returns a formatted string based upon the supplied arguments.
- string.gmatch() – when used in a loop, returns the next pattern match from the string.

- string.gsub() – searches and replaces all occurrences of the supplied pattern within a string.
- string.len() – returns the length of the supplied string.
- string.lower() – changes uppercase characters into lowercase characters.
- string.match() – returns a substring matching a supplied pattern.
- string.rep() – returns a string that includes  $n$  concatenated copies of the original string.
- string.reverse() – returns the reverse of the supplied string.
- string.sub() – returns a substring based upon character location.
- string.upper() – changes lowercase characters into uppercase characters.

If you want to make two strings into one string, you are performing what is referred to as concatenation. Concatenation is accomplished by using two dots ‘..’ between the objects to be concatenated.

For example, if I want to concatenate the words “My” and “house” into a single string, I would use the code:

```
local newString = "My" .. "house"
```

The only problem is that if I display that as text or print it to the terminal window, I get  
Myhouse

No space. If you want spaces included, you have to add them yourself.

Concatenation also works with variables or a mixture of variables and strings. Thus, if I wanted to create a string, I could:

```
local string1 = "My"  
local string2 = "house"  
local newString = string1 .. " " .. string2
```

would result in the string "My House"

## The Math API

You might be wondering that since we discussed the string API, what about the math API? The math API is extensive and we already used one of its components in the last chapter when we generated a random number to move the text on the screen. Here is a full list of

the available APIs in math. Each starts with the keyword math. In all examples x and y are a number:

- `math.abs(x)` – returns the absolute value of x.
- `math.acos(x)` – returns the arc cosine of x in radians (a number between 0 and pi). x must be between -1 and 1.
- `math.asin(x)` - returns the arc sine of x in radians (a number between -pi/2 and pi/2). x must be between -1 and 1.
- `math.atan(x)` - returns the arc tangent of x in radians (a number between -pi/2 and pi/2).
- `math.atan2(y, x)` - returns the arc tangent of y/x (in radians). Useful when converting rectangular coordinates to polar coordinates.
- `math.ceil(x)` – returns the smallest integer larger than or equal to x.
- `math.cos(x)` – returns the cosine of x in the range of -1 to 1.
- `math.cosh(x)` – returns the hyperbolic cosine of x.
- `math.deg(x)` – converts a radian value (x) to degrees.
- `math.exp(x)` – returns the value of  $e^x$ .
- `math.floor(x)` – returns the largest integer smaller than or equal to x.
- `math.fmod(x, y)` – returns the remainder of dividing x by y, rounding the quotient towards zero.
- `math.frexp(x)` – returns the split of x into a normalized fraction and an exponent.
- `math.huge` - returns a value larger than or equal to any other numerical value (basically an infinite number, but computers don't handle infinite very well, so a really, really big number).
- `math.inf` - same as math.huge, returns a value larger than or equal to any other numerical value.
- `math.ldexp(m, e)` – returns  $m * 2^e$ .
- `math.log(x)` – returns the natural logarithm of x.
- `math.log10(x)` – returns the base-10 logarithm of x.
- `math.max(x, ...)` – returns the largest value from the supplied arguments.
- `math.min(x, ...)` – returns the smallest value from the supplied arguments.
- `math.modf(x)` – returns the integer part of x and the fractional part of x.
- `math.pi` – returns pi.
- `math.pow(x, y)` – returns  $x^y$ .
- `math.rad(x)` – converts radians to an angle in degrees.
- `math.random([x][, y])` – returns a pseudo-random number. If x is not provided, then a number between 0 and 1 is generated. If x is provided, then a number between 1 and x. If x and y are provided, then a number between x and y is generated.

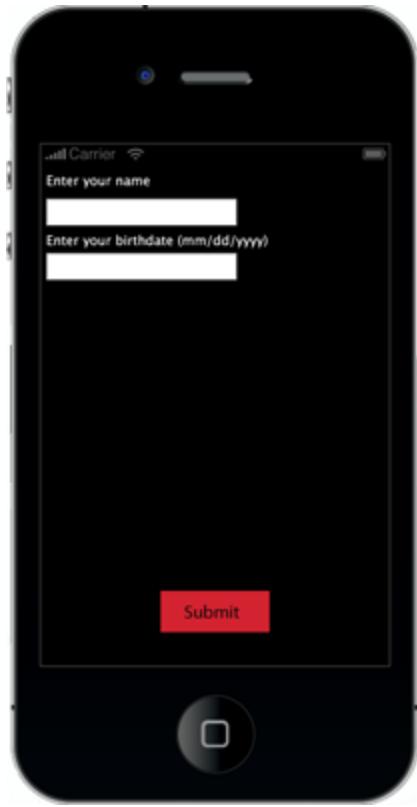
- math.randomseed(x) – sets x as the seed number for the pseudo-random number generator. If the same x is always used, the sequence of random numbers will be the same.
- math.round(x) – returns the x rounded to the nearest integer.
- math.sin(x) - returns the sine of x in the range of -1 to 1.
- math.sinh(x) - returns the hyperbolic sine of x.
- math.sqrt(x) – returns the square root of x.
- math.tan(x) - returns the tangent of x.
- math.tanh(x) – returns the hyperbolic tangent of x.

One last thing before we start our project. We need to setup our app so that if the user taps outside of TextFields, the keyboard will be dismissed. To handle this operation, I am going to use a background image that is 2048-by-1536. By using this size, I know that we will handle everything since 2048-by-1536 is the resolution of the iPad Retina.

### **Project 4.0: What's Your Age?**

For this project, we are going to build a fun little app that will ask for the user's name and birth date. Then we will do a few fun things to their name and compute how old they are in minutes and seconds. You will need two graphics for this project: the background image previously mentioned and a "Submit" button to start the calculations for age. I am targeting this for newer devices such as the iPhone 5 and the iPad Retina.

A warning before you get too far into this project. If you are developing on a Windows system, the TextFields and TextBox do not display in the Corona Simulator (at the time of this writing). If you are working on a Mac, everything will look and work fine. To make up for this little problem, after we go through the process of coding the app, we will walk through how to publish the app to an Apple or Android device.



After you have created a new folder and put the background image and a submit button in the folder, create a main.lua file. The first code we will add to our main.lua app is setup a host of variables to make our future calculations easier.

#### main.lua

```
local daysInYear = 365.2425 -- # days based on gregorian cal
local weeksInYear = daysInYear / 7
local daysInMonth = daysInYear / 12 --average # days a month
local weeksInMonth = daysInMonth / 7 -- # of weeks in a month
local secInMin = 60 -- # of seconds in a minute
local secInHour = 60 * secInMin -- # of seconds in a hour
local secInDay = 24 * secInHour -- # of seconds in a day
local secInWeek = 7 * secInDay -- # of seconds in a week
local secInMonth = daysInMonth *secInDay -- ave # of sec/month
local secInYear = daysInYear *secInDay --# seconds in a year
local todaysDate = os.date("*t")
--print (todaysDate.year, todaysDate.month, todaysDate.day)
```

Notice the last two lines of the code above. os.date is part of the API and returns the today's date (or whatever date the phone is set to). Using the "\*t" parameter tells os.date to return the current date and time as numbers: 2013, 02, 14, 12, 00, 00 would be the year 2013, second month, 14<sup>th</sup> day, 12<sup>th</sup> hour, 00 minutes, 00 seconds. You can see in the print statement that is commented out how to retrieve the specific information.

Next, we will load the background and store it as the variable background. By loading as the first object, we are making sure it is 'behind' everything else that is loaded to the screen. After loading the background, we will load the submit button.

```
-- load background to be used for dismissing keyboard
local background = display.newImage("bkgrd.png", 0, 0)

local submit = display.newImage("submit.png")
submit.x = display.contentWidth/2
submit.y = display.contentHeight-100
```

Now we will display instructions to the user and add the TextFields.

```
-- Get the users name and birthdate
local nameInstructions = display.newText("Enter your name", 10,
50, native.systemFont, 24 )

local usersName = native.newTextField(10, 100, 350, 50)

usersName.inputType = "default"

local bdayInstructions = display.newText("Enter your birthdate
(mm/dd/yyyy)", 10, 160, native.systemFont, 24)

local bday = native.newTextField(10, 200, 350, 50)

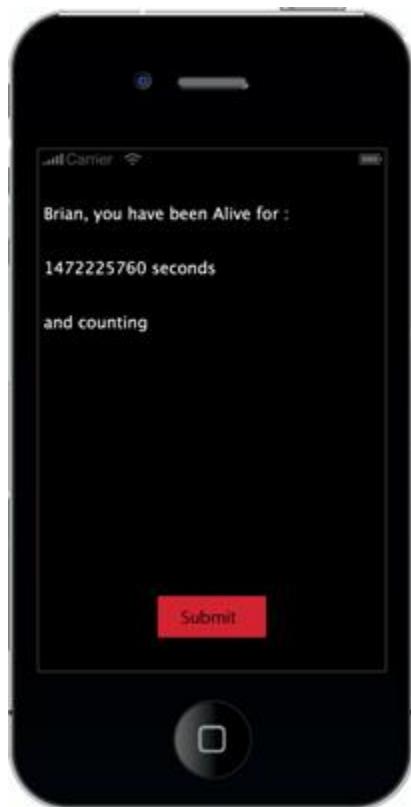
bday.inputType = "default"
```

Notice that we declare the type of keyboard that will be shown for each TextField. The next step is to setup the keyboard dismissal. This is accomplished by setting a tap event for the background that will call the function keyboardListener. The keyboardListener function removes the use of the keyboard by setting its focus to nil.

```
local function keyboardListener (event)
    native.setKeyboardFocus(nil)
end

background:addEventListener("tap", keyboardListener)

submit:addEventListener("tap", calculateAge)
```



Now we are ready to add the function that will be called by our submit button. The calculateAge function is going to make some heavy use of some of the String API functions that we have previously reviewed. I am adding this function above the background:addEventListener line and below keyboardListener function's end statement.

The first step in our function is to remove from the bday variable the year, month, and day of the user's birthday. At this time, we are going to assume that the information was entered in the correct form. To get the birth year, we will use two string functions: string.len and string.sub. It is possible to do it with just string.sub, but then we would miss out on all the fun of using both! String.len will return the length or number of characters contained in the birthday string. Notice that I am passing the parameter of bday.text. Anytime you are using a TextField or TextBox, you must use .text to work with what was stored in the variable.

```
local function calculateAge()

    -- Get the birth year, month, and day
    local bdayLen = string.len(bday.text)
```

Now that we have the number of characters that are stored, we can use .sub to extract a portion of the string. String.sub accepts a variable or string as the first parameter, the first letter position we want to extract as the second parameter, and the last letter position to be extracted as the final parameter.

```
local birthYear = string.sub(bday.text, bdayLen-3, bdayLen)
local birthMonth = string.sub(bday.text, 1,2)
local birthDay = string.sub(bday.text, 4, 5)
-- print(birthYear, birthMonth, birthDay)
```

Next, I would like to clear the screen of the previous text and TextFields. To do this, I am going to use a method called removeSelf(). RemoveSelf does just like it sounds it would do, it completely removes the object from the screen. Actually, it removes the object completely from memory. Poof! It no longer exists, nor can it be referenced again in the app! If I wanted the TextFields or text again, I would have to start from scratch and create them again.

```
bday:removeSelf()
usersName:removeSelf()
nameInstructions:removeSelf()
bdyInstructions:removeSelf()
```

Time to calculate some time! There are SO many ways that this could be done. I decided to go with a very straight forward approach. I calculated how many seconds since the date 1/1/1 for both current date and for the given birth date. After we have those two rather large numbers, we can subtract the birth date second count from today's date second

count, giving us the approximate number of seconds the person has been alive (give or take a few hours).

```
-- How many seconds from 1/1/1 to today
local totalSecToday = (todaysDate.year * secInYear) +
(todaysDate.month * secInMonth) + (todaysDate.day * secInDay)

--How many seconds from year 0 to birthdate
local totalSecBday = (birthYear * secInYear) + (birthMonth * secInMonth) + (birthDay * secInDay)

local totalSecAlive = totalSecToday - totalSecBday

local secAliveText = display.newText(usersName.text..", you have been Alive for :", 10, 100, native.systemFont, 30)

local secAlive = display.newText(totalSecAlive.." seconds", 10, 200, native.systemFont, 30)

local secAliveText2 = display.newText("and counting", 10, 300, native.systemFont, 30)
end
```

Finally, we can use a few `display.newText` lines to show the user's name, and how long they have been alive.

## Publishing to Device

First, as I have mentioned previously, 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. On a Windows system, you can only deploy to Android devices.

### Publishing to an Apple iOS Device

Apple regularly updates and changes their process for building to a device. For the latest build updates for Corona to an Apple device, check

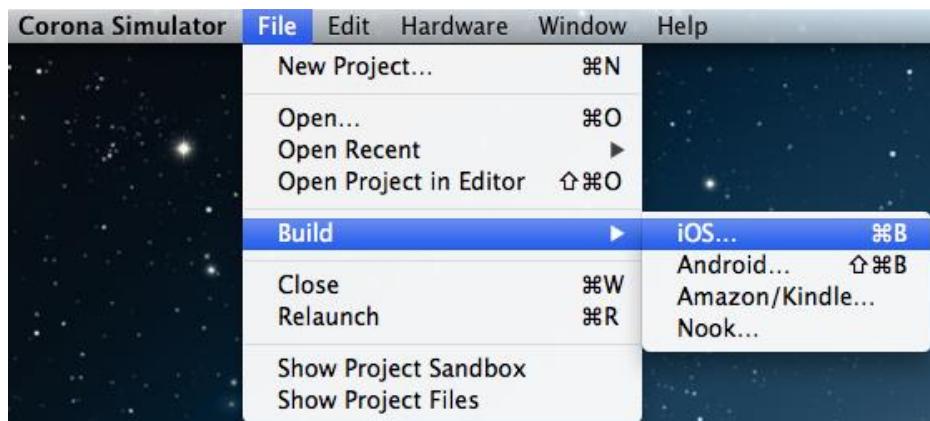
<http://developer.CoronaLabs.com/content/building-devices-iphoneipad>

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. If you need a walk-through of the provisioning process, one is provided in Appendix B.

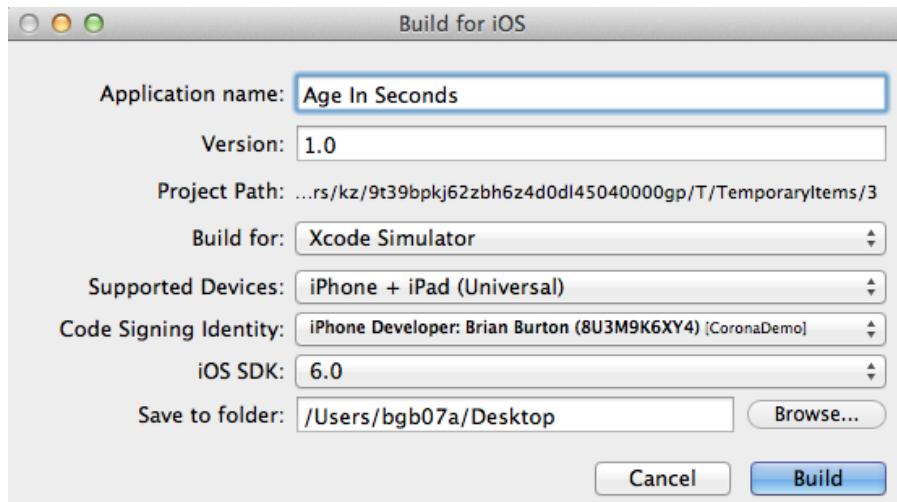
When building for an iOS device, you have the option of building to the iOS simulator (the simulator that comes with Xcode) or for a device. Building for the simulator provides you one more opportunity to get the bugs worked out before going through the time consuming process of deploying to a physical 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

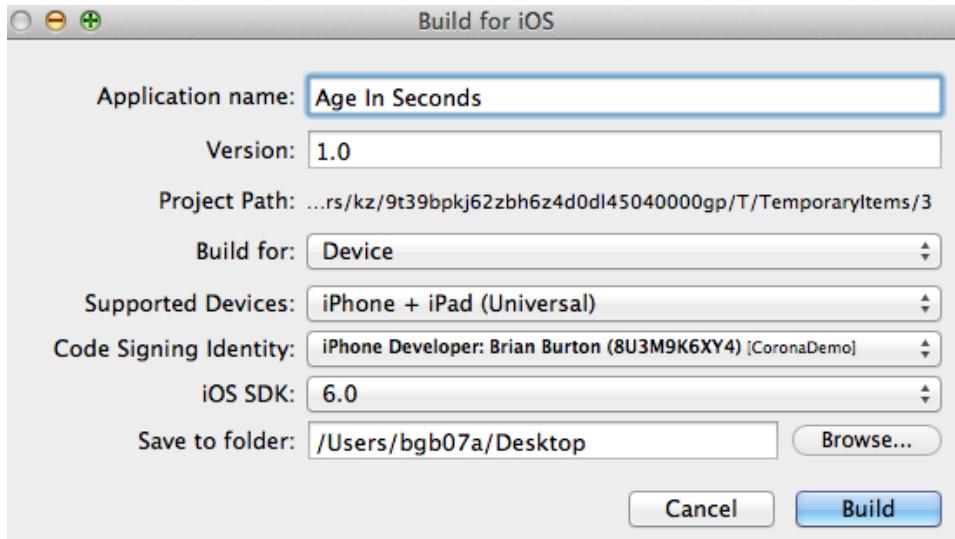


Verify your application name and version number, then change the 'Build for:' dropdown to Xcode Simulator instead of device. Change your supported Devices to iPhone only for this walk-through. Finally, select your Code Signing Identity. When you click on Build, the iOS Simulator will launch. Under the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

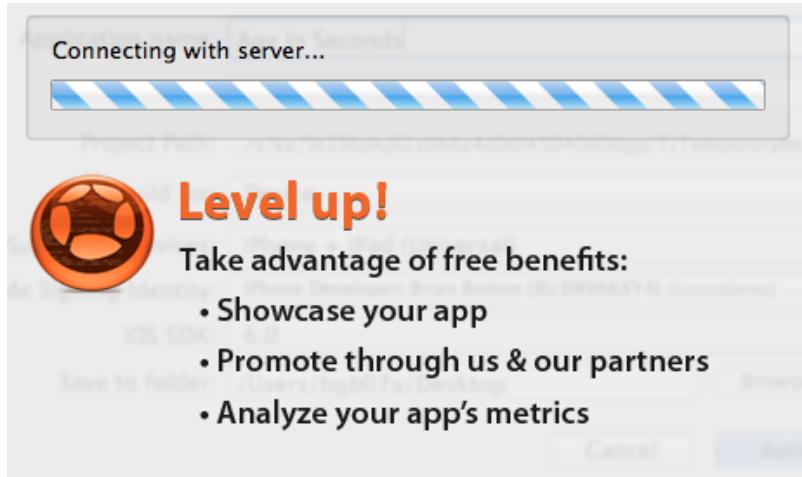
## Apple iOS Device Build

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. Go ahead and click build. You will get the following pop-up:

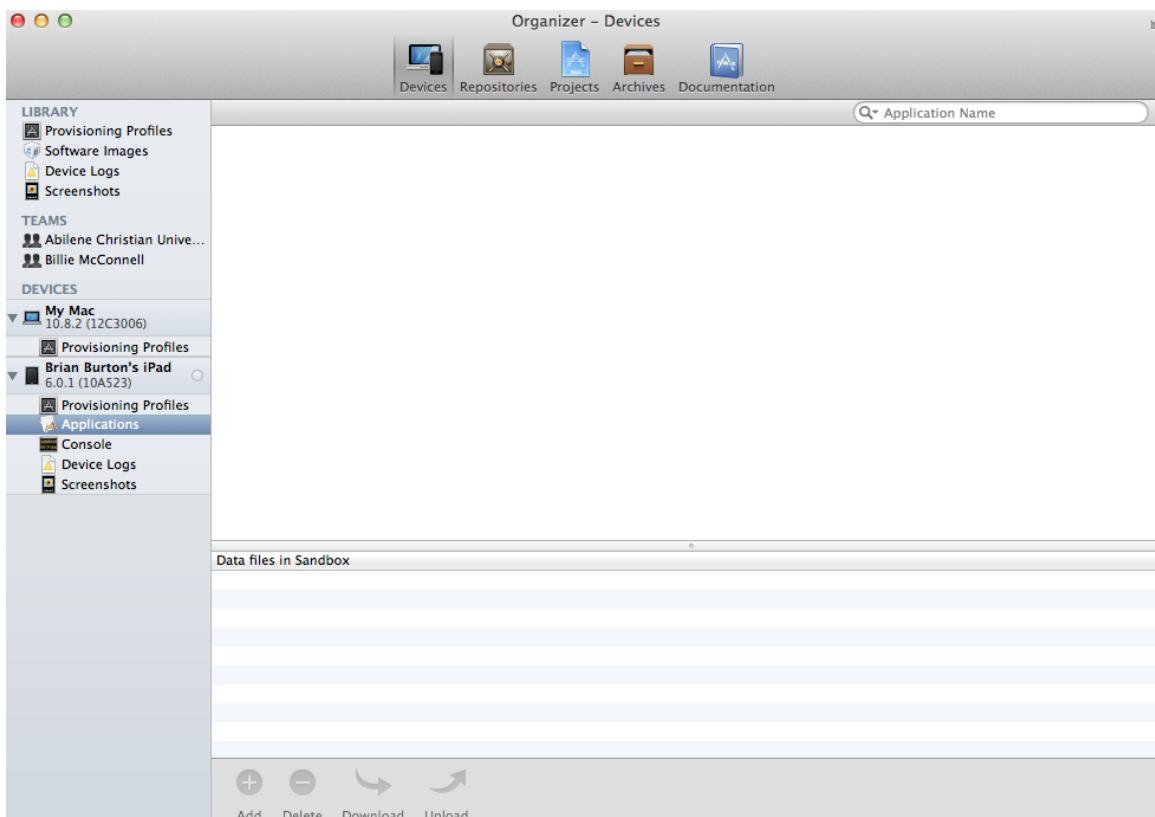


Once it finishes the build, you will probably receive a warning that the app will be accepted by Apple. This is because we did not include any icons for the app. We will discuss that further in the next chapter.

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

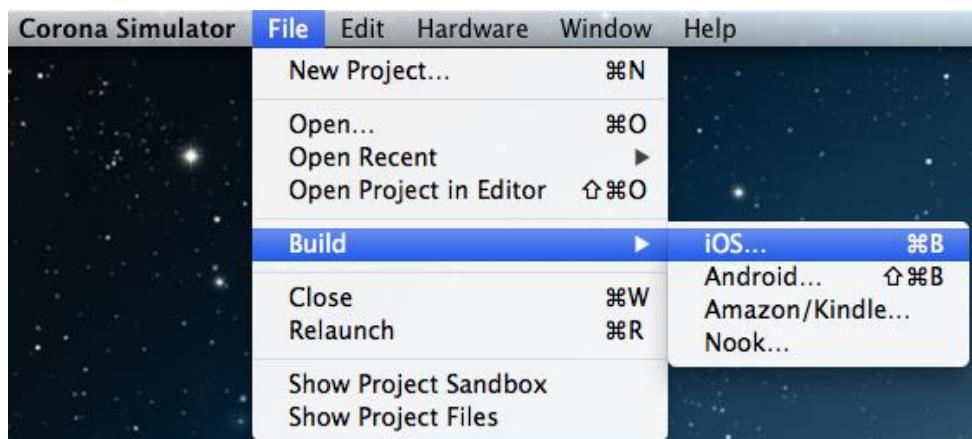


This will open the device organizer. If you have already configured your devices and provisions, you will be able to quickly add the app to your device. Under the device that you are using to test your app, select Applications. Then drag your build on the Organizer. This will install the app to your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



## Android OS Device Build

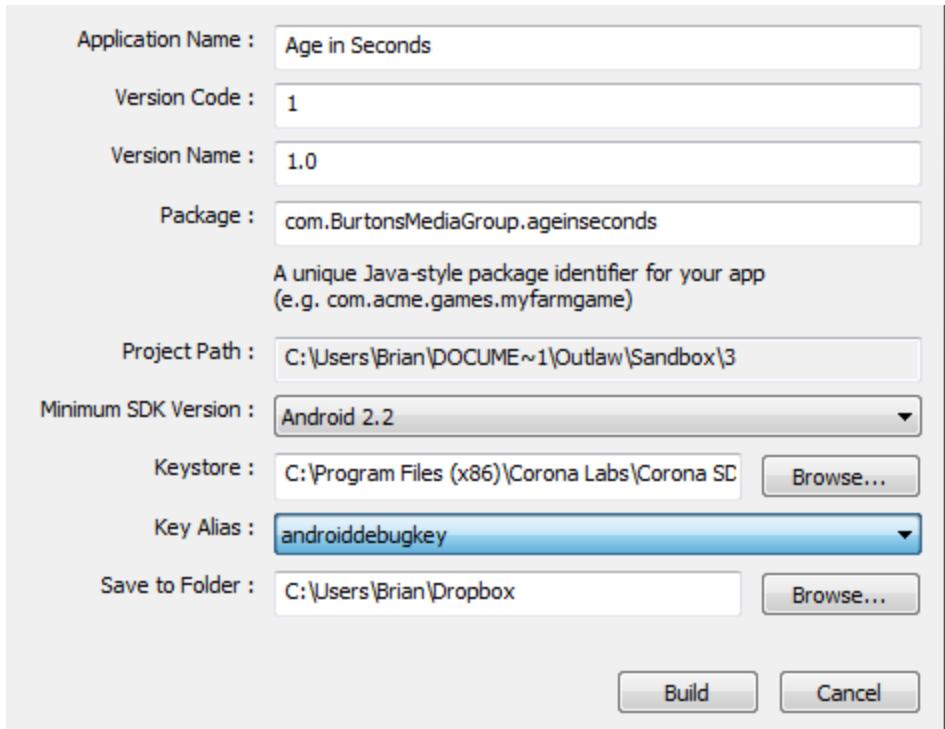
To build for an Android device, the device you are deploying to must have Android 2.2 or newer installed and have an ArmV7 processor. There are multiple ways to build for an Android device. I am going to cover the three most popular: command line, web server, and Dropbox. The first step is to prepare your device. Your tablet or phone must be set to debugging or development mode so that you can use it for testing. Go to your device's home screen, then press Menu> Settings> development and check the debugging box. Now we are ready 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.ageinseconds



Your target OS should already be set to Android 2.2 or newer. Make sure your Key Alias is set to androiddebugkey and click on Build. You may need to browse to the Corona Labs/Resources folder and select the debug.keystore. If the program asks for a password, use “android”.

Also, be sure that you have installed Java JDK 6. JDK 7 is not compatible with Android builds at the time of this writing. The link for downloading the current JDK (6.39) is <http://www.oracle.com/technetwork/java/javase/downloads/jdk6downloads-1902814.html>. Be sure to download the 32 bit version for Windows (Windows x86). This will create ageinseconds.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 ageinseconds.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.

**Note:**

Sometimes what we see in the simulator isn't what we get on the actual device. This is especially true with textfield and textbox on Android devices. If you reduce the font size and increase the textField height, the problem is usually resolved. If you are finding the text unreadable try the following code:

```
local inputFontSize = 18
local textHeight = 30
if (system.getInfo("platformName") = "Android") then
    inputFontSize = inputFontSize -4
    textHeight = textHeight + 10
end
myTextField = native.newTextField(200, 100, 200, textHeight)
myTextField.font = native.newFont(native.systemFont,
inputFontSize)
```

## Summary

Once again we have covered a lot of material very quickly! I recommend that you spend a little time with the string and math API functions. They are very powerful and when combined with loops and if-then, you can do a lot of very powerful calculations! I also recommend that you do practice deploying to your test device(s) from this point forward. It is very important to get an idea of how your app will look on an actual device.

## Programming Vocabulary

Concatenation

Strings

## Questions

1. Define concatenation.
2. Using the math API, how can you find the value of PI?
3. What is the difference between a textfield and a textbox?
4. Which keyboard should be used to enter a phone number? for a URL?
5. How can the keyboard be dismissed?
6. Explain one of the methods for deploying to an Android device.
7. Name one of the required things you must have to deploy to an Apple device.
8. Which property contains what a user typed in a textfield?
9. To reverse a string, use the which string API commands?
10. Can a textfield hide passwords entered into an app?

## Assignments

1. Revise Project 4.0 to also show how many days and months the person has been alive.
2. Create an app that shows how long it will be until the user's next birthday.
3. Create an app that asks for the users name then shows the users name with the letters in reverse order.
4. Create an app that asks for the users name and then displays the name down the side of the screen, one letter at a time.
5. Challenge: Using if-then statements, check to make sure that the user entered the birth date information correctly.

# Chapter 5 Working with Graphics

## Learning Objectives

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
- Identify App Store Icon requirements
- Explore additional animation methods

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 and creating animation frame by frame.

## 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 graphic API commands:

- `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:setStrokeColor(r, g, b [,a]) - Sets the color of a line object based upon r, g, b (and optionally alpha) values between 0 and 1
- object:setFillColor(r, g, b[, a]) - Sets the fill color for vector objects based upon r, g, b (and optionally alpha) between 0 and 1
- display.setDefault(r, g, b) - Allows you to change the default color (white) to another fill color.

display.newLine has a few of its own properties:

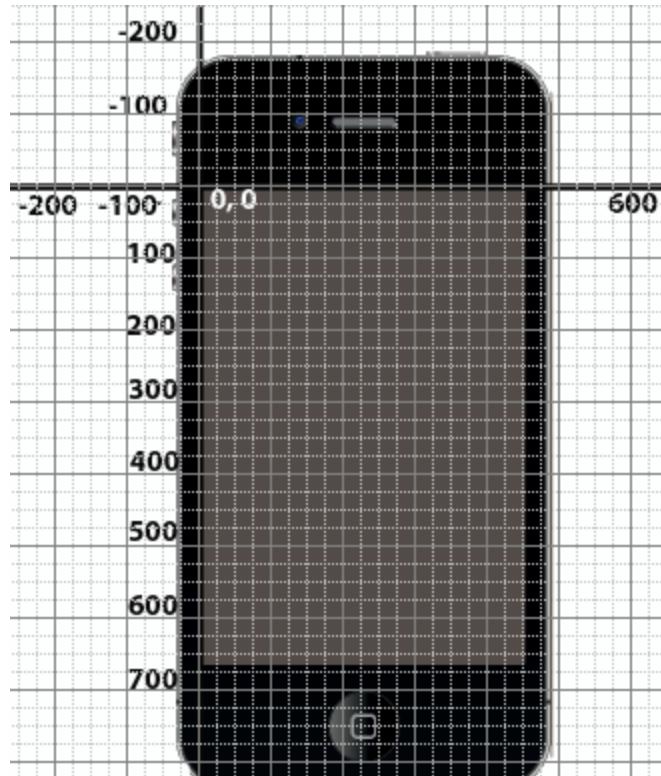
- object:append() - Appends one or more line segments to an existing display.newLine object
- object.width(x) – sets the stroke width of the line in pixels. Can also be used to find the width of an object

Now that you have seen the commands, let's put them into use.

## 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. To get started, create a folder and main.lua file to start.

As you think about the coordinates on the mobile device, remember that the left, top corner is coordinate 0, 0. Anything above or to the left of the top left corner (i.e, off of the screen) would be a negative coordinate. By using negative coordinates or coordinates that are greater than the resolution of the device, we can slide items onto the screen from the sides. You can think of the drawing space on a mobile device as a flipped Cartesian coordinate system:



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 stroke color to white and the stroke width to 3 pixels.

### **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:setStrokeColor( 1, 1, 1, 1 )
star.strokeWidth = 3
```

As you might have noticed, a portion of the star is off the screen (the negative numbers), 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( 1, 0, 0)
rectangle:setStrokeColor(0, 0, 1)

local circle = display.newCircle(display.contentWidth/2,
display.contentHeight/2, 15)
circle.strokeWidth = 2
circle:setFillColor(0, 1, 0)
circle:setStrokeColor(1,1,1)
```

To create a vector rectangle object we use **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.

To correct this problem, we can either move the object by subtracting 110 pixels form the y parameter or we use the .anchorY parameter. Changing anchorY 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. By default the anchor point for any object is the center of the object. But as this object is multiple line segments, it is necessary to reset the anchor to the center of the object.

The anchorX and anchorY parameters range from 0 to 1, with 0 being the left (x), or top (y), and 1 being the right (x), bottom (y). By default both anchorX and anchorY are set at .50 (the center of the object). Let's reset the star's anchor with

```
star.anchorY = .5
```

before the transition.to commands and run your app to see the difference. Being able to change the x and y reference point on an object can be very handy if you need to rotate the object on an edge or corner instead of on the center of the object.

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:setStrokeColor( 1, 1, 1, 1 )
star.strokeWidth = 3
star.anchorY = .5

local rectangle = display.newRect( 100, 100, 50, 50)
rectangle.strokeWidth = 5
rectangle:setFillColor( 1, 0, 0)
rectangle:setStrokeColor(0, 0, 1)

local circle = display.newCircle(w, h, 15)
circle.strokeWidth = 2
circle:setFillColor(0, 1, 0)
circle:setStrokeColor(1,1,1)

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? Sure, it has been done (I am thinking of several classic arcade and early home console games from the 70's and 80's such as Lunar Lander or Star Fox), 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. Any photo that you take with your smartphone or digital camera is an example of a bitmap graphic. Generally they are saved as either jpg or png. PNG is the recommended bitmap format for maximum compatibility across multiple mobile device 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 plan to export. 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.

## Icons

While we are on the topic of resolution, 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 the minimum 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 & 5+)	114x114px
Icon-72.png (iPad 1 & 2)	72x72px
Icon-144.png (iPad 3+)	144x144px

Oh, don't forget you will need a 1024x1024px of the icon for the iTunes store. To make life easy on yourself, start with the 1024x1024 resolution of your icon, then resize the image to the required, smaller, sizes.

A great online tool for making your icons and splash screen is:

<http://www.appicongenerator.com/>

## **Build.Settings and Config.lua**

To handle all of the many different devices that are available, Corona has two files that help instruct the SDK on how to handle various device builds or configurations that might occur when you deploy your app to many different types of devices.

### **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 the same folder as your main.lua file.

Sample build.settings:

```
settings =
{
    orientation =
    {
        default = "portrait",
        supported =
        {
            "portrait", "portraitUpsideDown", "landscapeRight",
"landscapeLeft"
        }
    },
}
```

In this sample build.settings file, we configure the default orientation to portrait and also support 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

As we have previously discussed, not all mobile devices have the same resolution. The working screen resolution for early iPhones and iPods 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 or load different resolutions of images 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 the same folder as your main.lua.

## Dynamic Content Scaling

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.

Sample config.lua file:

```
application =
{
    content =
    {
        width = 320,
        height = 480,
        scale = "letterbox"
    },
}
```

## Dynamic Image Resolution

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 (starting with the iPhone 4) by adding "@2" suffix to their filenames.

Corona uses a more general method for defining alternative images 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 (isn't that nice of them?).

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

With this example 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.

To load your images, use

`display.newImageRect(filename, base width, base height)`

and Corona will choose the closest matching suffix, as defined by your scale.

## 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
```

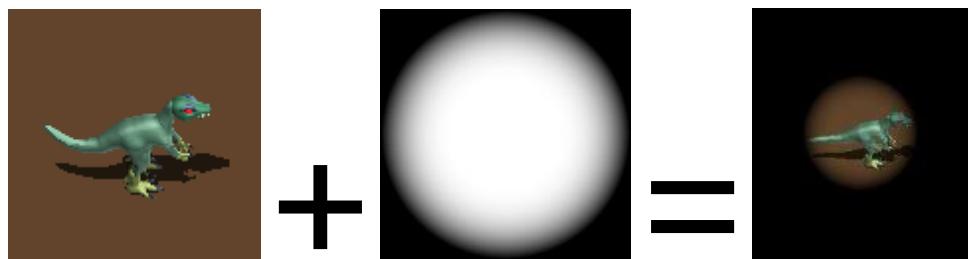
```
myImage:scale(.5, .5)
```

This short code would result in an image that was displayed at  $\frac{1}{4}$  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:  $.5 \times .5 = .25$ . 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 (which will be 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 mask 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
```

### Project 5.1: Masks

This is a short little app to give you a quick idea of how masks work. I have created 2 images, the first a simple graphic with black and white bars, the second with the text “Corona Rocks.”

First, in the `build.settings` file, we will set the default orientation of the app to landscape right.

```
build.settings
```

```
settings = {  
    orientation =  
    {  
        default = "landscapeRight",  
    },  
}
```

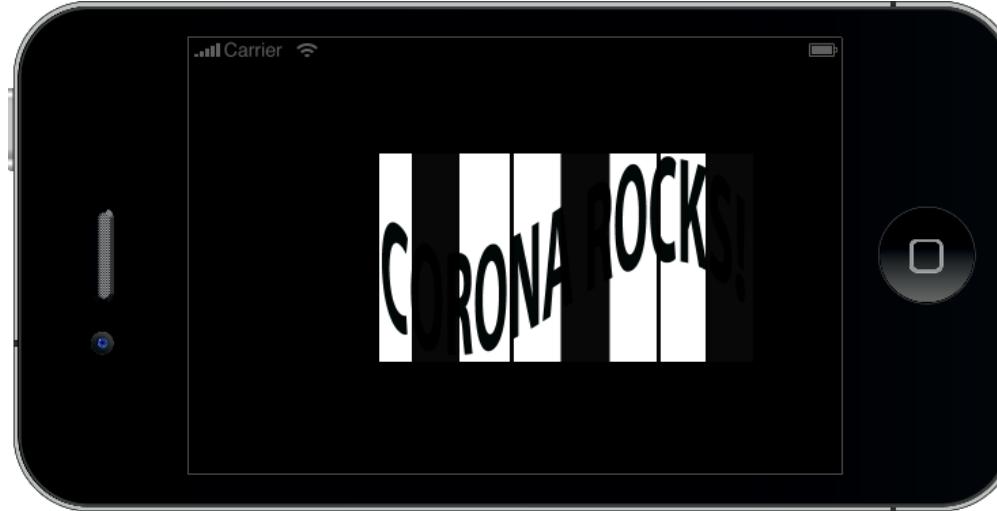
We are going to load the “Corona Rocks” image and apply the bars as a mask, which will cause some of the characters to look faded.

```
main.lua
```

```
local corona = display.newImage("corona.png",  
(display.contentWidth/2)-200, (display.contentHeight/2)-150 )
```

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

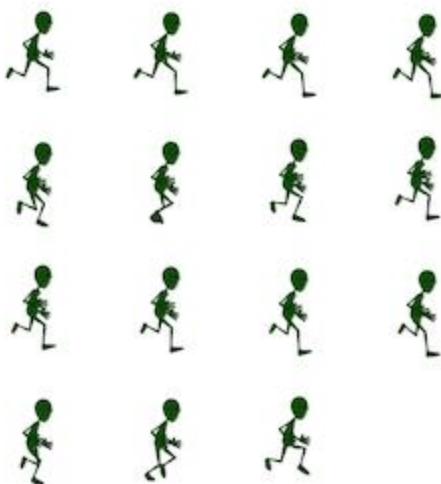
And now we have a simple mask applied to our image!



## Sprite Sheets

Sprite sheets are commonly used for games and animation. They are a series of 2D images saved in a single png image file (which is called a sprite sheet). This allows for a more effective and efficient use of memory. Individually, each of these images might use 5 or 10k of memory. While not a problem for just a few images, it quickly adds up if we have a significant number. By placing them all in one image file, we use much less device memory, which helps game and app performance. We can also take advantage of their placement on the sheet to simulate movement by displaying the images in the same location one after another.

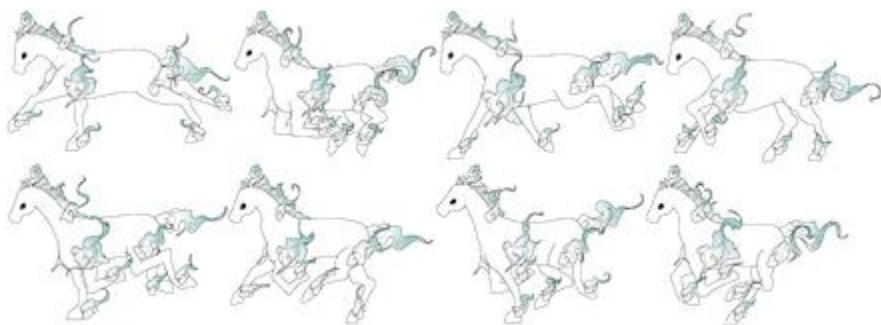
Corona provides support for two types of sprite sheets: uniform frames and non-uniform frames. Uniform frames are 2D images that are all the same size throughout the sprite sheet. The image below is an example of a uniform frame sprite sheet provided in the Jungle Scene sample project that ships with 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 of sprite sheet, non-uniform frame, contains 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 (called a sprite data sheet) 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 Texture Packer, Amino, Sprite Helper (mac only), and Zwoptex (mac only). More information is provided on each of these programs in chapter 20. If you would like to learn more about the process of creation of sprite sheets, I recommend Andreas Loew (developer of Texture Packer), who has created a great video on the subject:

<http://www.codeandweb.com/what-is-a-sprite-sheet>. Since how to create the sprite sheet

is a little outside the scope of this book, we are going to focus on using a few already made sprite sheets.

Sprites are incredibly useful and powerful for game creation. We will examine how to create a game that makes use of sprite sheets in a later chapter. For now, we will take what we have learned about sprite sheets to create a simple project.

## Project 5.2: 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 (our sprite data sheet). 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>). I used SpriteHelper to create the sprite sheet from bmp files and to create the animation sequences.



### *Uniform space and size sprite sheet*

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

You will first need to load the sprite sheet into memory. To do this, we need to tell Corona how the sprite sheet is laid out. We can do this by hand by giving all the dimensions, which would look something like:

```
local options = {  
    width = 128,  
    height = 128,
```

```
    numFrames = 87  
}
```

Or we can take advantage of the fact that we used a program like SpriteHelper to create our sprite sheet. SpriteHelper (as well as other sprite sheet tools) allows for the creation of a lua file that will pass all of the parameters of our sprite sheet. To take advantage of the external lua file, we first have to load it into memory:

```
local GreenDinoSheetData = require("GreenDinoSheet")
```

The key word “require” tells Corona that this is an external program file with functions and information that will be used in the app. There is no need to include the .lua file extension. Corona understands that this will be a .lua file. Now all we need to do is load the data from the sprite data file:

```
local options = GreenDinoSheetData.getSpriteSheetData()
```

The command GreenDinoSheetData.getSpriteSheetData() is a function call. By putting GreenDinoSheetData first, Corona understands that this function will be located in the GreenDinoSheet.lua file that we required. The function name is getSpriteSheetData. By assigning it to the variable options, we are telling Corona to store all of the data that is in the getSpriteSheetData function in the options.

Those two lines of code do all of the heavy lifting for us, and we are now ready to load graphics on the screen:

```
local sheet1 = graphics.newImageSheet("GreenDinoSheet.png",  
options)  
local GreenDino = display.newImage(sheet1, 2)  
GreenDino.x = display.contentWidth/2  
GreenDino.y = display.contentHeight/2
```

We are using a new command here: graphics.newImageSheet. We are assigning the sprite sheet to the variable sheet1, passing it the name of the sprite sheet and the options variable. Corona now knows all about the sprite sheet: that each of the images is 128 x 128, that there are 87 of them, and how to access each one by number. The first image in the file is image 1, the second is 2, etc.

Thus, if I want to display an image from the sprite sheet on the screen, I can use display.newImage and pass it sheet1 and the image number I want to load on to the screen (just like I did above).

Try it out. Also change the number 2 above to any number between 1 and 87 to see different images.

I know, you want to see animation! Where is the rampaging dinosaur!?! Let's work with that next.

### Project 5.3 Sprite Animation

SpriteHelper (and most other sprite sheet tools) has another neat feature that lets me create animation sequences from the sprite sheet, just like the GreenDinoSheet.lua file, these sequences allow me to easily animate my dinosaur. In this project, we are going to make the dinosaur appear to walk around on the screen.

If you open the Walking.lua file you will find the following information:

#### Walking.lua

```
function getAnimationSequences ()
    local sequences = {
        name = "Walking",
        frames = {80,81,82,83,84,85,86,87},
        time = 800,
        loopCount = 10,
    }
    return sequences
end
```

SpriteHelper again did most of the hard work for me. It provides Corona with the basic information for a walking sequence and even names it Walking. The walking images in the sprite sheet are images 80 through 87. It also provides a basic timing loop for how quickly to play the images to make it look more 'realistic'.

Let's take advantage of this information to create a simple animation. To begin with, we will load the external lua files that contain our data. As you can see, we are including WalkingData to load the file and Walk to load the specific walk sequence.

## main.lua

```
local GreenDinoSheetData = require("GreenDinoSheet")
local WalkingData = require("Walking")

local options = GreenDinoSheetData.getSpriteSheetData()
local Walk = WalkingData.getAnimationSequences()
```

Next, we will load the sprite sheet with the `graphics.newImageSheet` command. Then we will load the animation sequence with the command `display.newSprite`. It is possible to group your animation sequence into one file, but for simplicity, I broke it into multiple animation sequences.

```
local sheet1 = graphics.newImageSheet("GreenDinoSheet.png",
options)
local dinoWalk = display.newSprite(sheet1, Walk)
```

Now we will place the `dinoWalk` animation on the screen and tell it to play (which will start the animation).

```
dinoWalk.x = 64
dinoWalk.y= display.contentHeight/2

dinoWalk:play()
```

Hmm, needs one more thing, don't you think? Wouldn't it be nice if the dino actually walked?

```
transition.to(dinoWalk, {x=display.contentWidth, time= 10000})
```

Here is list of the various sprite commands and what they do:

- `graphics.newImageSheet(image, layout)` – creates the sprite sheet given the image file and the layout of the sprites in the image file.
- `display.newSprite(imagesheet, animation sequence)` – create a sprite that can be played.

## Sprite Control Methods

- `animation:play()` – starts the animation created with the `display.newSprite` command.
- `animation:pause()` – pauses an animation that is playing.
- `animation:setFrame(frame)` – sets the start frame of the animation.

- `animation:setSequence(sequence)` – changes to a different sequence if you have multiple sequences loaded.

## Sprite Properties

- `object.frame` - returns the current frame of the animation.
- `object.isPlaying` – returns true if the animation is currently playing.
- `object.numFrames` – returns the number of frames in the current animation sequence.
- `object.sequence` – returns the name of the currently playing animation sequence.
- `object.timeScale` – will return or set the scale to be applied to the animation time. Can be used to speed-up or slow-down an animation by a multiple of the value given. Minimum of 0.05, maximum of 20.0.

## Sprite Event Listeners

Sprite event listeners are an easy way to check on the status of your sprite. Using an event listener, you can cause your animation to move to the next animation or determine when it has finished the current animation sequence. We will discuss event phases more in chapter 9, but here is what you can listen for with sprites:

- began – the sprite has started playing.
- ended – the sprite has finished playing.
- bounce – the sprite bounded from forward to backward while playing (bounce is also a play setting).
- loop – the sprite has looped to the beginning of its sequence.
- next – the sprite has moved on to next sequence.

## Other Uses of Image/Sprite Sheets

If you have thought about it, Image sheets have far more uses than just loading graphics. For example, if you needed to include a special looking font or characters in your app, this is an easy way to import and use the graphics to get a very special look and feel. If you have ever tried to create your own font, you will appreciate this use.

Note: If you use SpriteHelper for a similar project, I had to modify all of the animation files, removing the title of the animation from within the file. If you make the file look similar to what is above, yours should work.

## 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,

loading a bitmap image, 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.

## Questions

1. Which type of graphic uses less memory and is infinitely scalable?
2. Can vector graphics be moved once they are placed on the display?
3. What are the four icon sizes for an iOS app?
4. What are the three icon sizes for Android apps?
5. What is the maximum size of a bitmap image in Corona?
6. Describe the function of a mask.
7. To animate a sprite, what information must you have?
8. Name one use of the config.lua file.
9. Name one use of the build.settings file.
10. What is the naming convention for bitmap graphics that are at different resolutions?

## 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 sheet of green dino, create a short dramatization. Included in the green dino sheet are animation sequences for look, roar, run, and walk.
- 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.

# Chapter 6: Creating the User Experience

## Learning Objectives

In this chapter we will be learning about creating a good user experience. This includes learning:

- How to hide and show the status bar
- How to load custom fonts
- How to group objects
- How to create and use external modules
- How to use composer API to create additional views for the user

## The User Experience

One of the goals of creating a good mobile application or game is to create an enjoyable user experience. The user should be able to easily navigate and find the resources that they are looking for in your app. They should not need to navigate through multiple screens to accomplish the goal of your apps. Requiring the user to navigate through more than two screens (sometimes also referred to as views) is signs of a poor user interface.

Creating a good interface is the first step toward creating a good user experience. People who are serious about creating successful apps consider multiple design configurations and even the look of the icon. I recently read where a very successful small company that makes mobile games designed over 30 icons for their game and then play-tested their design and icons to ensure that they had the best possible design that their target audience would find appealing.

A few things to consider when developing your app include:

- Group objects that perform similar functions or are a part of the same function. Placing a rectangle around the functions can help inform your user that they are part of the same functionality.
- Use color to inform your users of functionality. Using red to signify stop and green to go is very helpful to those who will use your app.
- Use a minimum of 11pt font for readability.
- Be careful with your selection of a font type. Some fonts are built in to the mobile device, while others will be replaced with default fonts.

- Utilize buttons to make actions and navigation options clear to those who will use your app.

## 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 or tablet 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 – returns status bar to normal or default setting
- display.TranslucentStatusBar – sets the status bar to translucent
- display.DarkStatusBar – sets the status bar to dark.

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.

## Custom Fonts

When working with multiple types of devices running on different operating systems, it can be very difficult to know if a particular font is available on the system. For most applications, using the default font will be sufficient. However, there are times when the judicious use of custom fonts can be a great way to improve your app. You can find many free fonts at <http://www.1001freefonts.com/> or <http://www.dafont.com/>. Be sure to check the `readme.txt` file included with the font if you will be using your app for commercial purposes (i.e. you plan to make money with it in any way).

Custom fonts can be easily incorporated into applications as long as they are registered in the `build.settings` file.

```

settings =
{
    iphone =
    {
        plist =
        {
            UIAppFonts =
            {
                fontFileName
            }
        }
    }
}

```

For Android, simply including the font in the application folder is sufficient. You must also have the exact font name (not just the name of the file containing the font) for the display.newText API call.

## Project 6.0 Custom Fonts

Our goal for this project is to load a custom font and display it to the screen. To accomplish this we are going to learn a few more commands and procedures. The method of using custom fonts on an Apple and Android device vary slightly, but I will show you a great work around that will make it easy!

First we will need to configure the build.settings file:

build.settings

```

settings =
{
    iphone =
    {
        plist =
        {
            UIAppFonts = {"Baroque Script.ttf"}
        },
    }
}

```

If the BaroqueScript.ttf file is located in the same folder as the main.lua and build.settings file, it will be loaded for use.

Next, we will determine what the font is named inside the font file. The internal name can vary widely, so it is a good idea to perform this procedure anytime you are using a custom font. **IF you are running this on the Corona Simulator, you will need to make sure the font is installed before you run the simulator.** If you don't install it first, it will not show up on the list. This app will go through and show the fonts that are currently loaded in the system memory. If your font isn't installed, it won't show up.

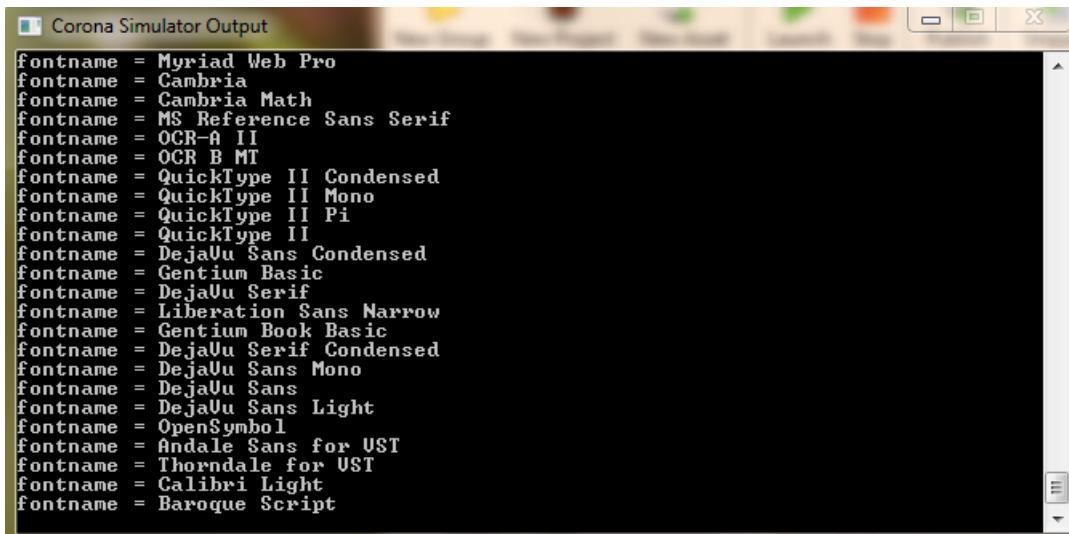
The first thing we will do is load the native fonts into a variable called 'font'.

```
local fonts = native.getFontNames()
```

Next we will display the font names that are available using a 'for do' command. This command allows us to look at information that is installed in an array or table (which we will discuss more in chapter 10). For now, what you need to understand is that the command will step through all of the fonts that are loaded in and display them to the terminal window.

```
-- Display each font name in the terminal console
for i, fontname in ipairs(fonts) do
    print( "fontname = " .. tostring( fontname ) )
end
```

At the end of the list you should see the font that you loaded in the build.settings file.



A screenshot of a Windows-style application window titled "Corona Simulator Output". The window contains a list of font names, each preceded by the word "fontname =". The list includes standard system fonts like Myriad Web Pro, Cambria, and Arial, as well as the custom font "Baroque Script" which was loaded from the build.settings file.

```
fontname = Myriad Web Pro
fontname = Cambria
fontname = Cambria Math
fontname = MS Reference Sans Serif
fontname = OCR-A II
fontname = OCR B MT
fontname = QuickType II Condensed
fontname = QuickType II Mono
fontname = QuickType II Pi
fontname = QuickType II
fontname = DejaVu Sans Condensed
fontname = Gentium Basic
fontname = DejaVu Serif
fontname = Liberation Sans Narrow
fontname = Gentium Book Basic
fontname = DejaVu Serif Condensed
fontname = DejaVu Sans Mono
fontname = DejaVu Sans
fontname = DejaVu Sans Light
fontname = OpenSymbol
fontname = Andale Sans for UST
fontname = Thorndale for UST
fontname = Calibri Light
fontname = Baroque Script
```

As you can see in the image above, "Baroque Script" is at the bottom of my list. Now comes the fun part. Since Android and Apple handle loading fonts differently, I can GREATLY simplify my life by just changing the filename in the folder to match the name of the font that is listed above!

The next step then is to change the filename to “Baroque Script.ttf” in my project folder. After changing the file name, you will also need to modify it in your build.settings file:

#### build.settings

```
settings =
{
    iphone =
    {
        plist =
        {
            UIAppFonts = {
                "Baroque Script.ttf"
            }
        },
    },
}
```

Note that you may need to reinstall the font to your Windows or Macintosh since the file name has changed to system to continue testing in your Corona Simulator.

One last line to add, the display.newText that shows the font in use:

```
display.newText("Hello World", 10, 10, "Baroque Script", 24)
```

You can add up to 100 fonts in this way. Just place a comma between each font in the build.settings file.



## 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 6.1: Group Movement

In this project we are going to load three images (a couple of button images), 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 screens or views of an app without the need to do all that extra programming. 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 functions and code segments are used all the time. Fortunately, Lua allows us to create modules that can be loaded and reused in our apps quickly and easily. The use of additional Lua files is best shown when you begin to use the Composer API.

## Composer

Composer supplies built-in scene (also referred to as views or screens) creation and management. This means that you can create multiple views or screens and be able to easily load and unload them as you need. Composer includes easy transition methods between individual scenes. The scene object is an event listener that responds to specific events, simplifying scene management for your applications.

## Project 6.2 A Simple Story

For this first project using composer, we are going to create a simple project that uses 3 Lua files for different views or screens. We will have a main.lua, mom.lua, and dad.lua.

To get started, we will first write our main.lua file. Remember to place all of your lua files and graphics in the same folder. We only need two commands to get everything started using composer. The first is to load composer into memory using the require "composer" command. Second, we will use composer to transition to the mom.lua file using a fade transition effect that will take 400 milliseconds to perform.

main.lua

```
local composer = require "composer"

-- Load the first scene to be shown
composer.gotoScene( "mom", {effect= "fade", time = 400} )
```

The gotoScene command API is:

`composer.gotoScene(sceneName [, options])`

where *sceneName* is the name of the lua file to be loaded and run. Options include the transition effect to be used and how long the transition should take to perform. Transition effects include: fade, zoomOutIn, zoomOutInFade, zoomInOut, zoomInOutFade, flip, flipFadeOutIn, zoomOutInRotate, zoomOutInFadeRotate, zoomInOutRotate, zoomInOutFadeRotate, fromRight (over original scene), fromLeft (over original scene), fromTop (over original scene), fromBottom (over original scene), slideLeft (pushes original scene), slideRight (pushes original scene), slideDown (pushes original scene), slideUp (pushes original scene), and crossfade.

You can also pass additional parameters as part of the transition; a nice way to pass variables that might be used in the next segment of your program! Now that we have called mom.lua from main.lua, we should probably say hi! We will start the mom.lua file with a comment to help us remember which file we are working in and load the composer api into memory (this will need to be done for each scene or file that uses composer). Next we will create a variable called *scene* to hold all of the information about this scene. Finally, we will create a display group as discussed earlier in the chapter to make viewing the scene information easier.



mom.lua

```
-----  
- mom.lua  
-----  
  
local composer = require "composer"  
  
local scene = composer.newScene()
```

After we setup our variables, we will setup the scene like we would normally if we were just working with only a main.lua file with a few minor variations. Composer is incredibly efficient in making sure that the scene can be pre-loaded, shown multiple times in a sequence, hidden when the user leaves the scene, or disposed of to free memory.

To take advantage of this efficiency, we will need to use a few special functions to manage loading and unloading the scene. The first is the **create** function. This function preloads all of the information before showing the scene. This can make for very efficient memory usage. **Create** will only execute if it isn't already loaded into memory. If the scene has already been visited, it will not execute.

Thus, as we are designing our scene, we need to keep in mind that it might be loaded and hidden from view several times. This is where our display group becomes very useful. By adding all of our display objects to a display group, we are able to hide or show all of them at once by changing the alpha. To begin with, we will just load the images and text then setup an event listener should the background be tapped.

```
-----  
- BEGINNING OF YOUR SCENE  
-----  
  
--Called if the scene hasn't been previously seen  
function scene:create ( event )
```

```

local MomGroup = self.view

bgImage = display.newImage("bg.jpg", 0, 0)
MomGroup:insert(bgImage)

hiMomText = display.newText("Hi Mom!!", 0, 0, nil, 36)
hiMomText.x = display.contentWidth/2
hiMomText.y = display.contentHeight/2
MomGroup:insert(hiMomText)

end

```

The `scene:createScene` function will handle loading our graphics and text to the screen. Next we will create a little function that when the user touches the background, we can fade to the `dad.lua` file over 400 milliseconds.

```

function onBackgroundTouch()
    composer.gotoScene("dad", "fade", 400)
end

```

Now we need to handle the situation of the user coming back to this scene after it has been hidden from view. Using the `scene:show` function, we will initialize a event listener to handle the background touch (which will call the fade to dad routine).

```

function scene:show(event)
    bgImage:addEventListener("touch", onBackgroundTouch)
end

```

Now we just need to remove the event listener when composer determines we are leaving (or exiting) this scene. This is handled by the `scene:hide` function:

```

function scene:hide(event)
    bgImage:removeEventListener("touch", onBackgroundTouch)
end

```

All that is left is to setup our event listeners and give a final required command when using the composer API: return scene. Return scene communicates back to the `main.lua` file that everything has been loaded so that composer can continue running properly.

```

-- "create" is called whenever the scene is FIRST called
scene:addEventListener("create", scene)

-- "show" event is dispatched whenever scene transition has

```

```

finished
scene:addEventListener( "show", scene )

-- "hide" event is dispatched before next scene's transition
begins
scene:addEventListener( "hide", scene )

return scene

```

Now that we have mom.lua taken care of, we will do basically the same thing with dad.lua with the exception that dad.lua will pass control back to mom.lua should the background be tapped.



dad.lua

```

-----
- dad.lua
-----

local composer = require( "composer" )
local scene = composer.newScene()

-----
- BEGINNING OF YOUR SCENE
-----

--Called if the scene hasn't been previously seen
function scene:create( event )

```

```

local DadGroup = self.view

backgroundImage = display.newImage("bg2.jpg", 0, 0)
DadGroup:insert(backgroundImage)

hiDadText = display.newText("Hi Dad!", 0, 0, nil, 36)
hiDadText.x = display.contentWidth/2
hiDadText.y = display.contentHeight/2
DadGroup:insert(hiDadText)

end

local function onBackgroundTouch()
    composer.gotoScene( "mom", {effect = "zoomInOut", time = 800}
} )
end

function scene:show(event)
    backgroundImage:addEventListener("touch", onBackgroundTouch)
end

function scene:hide(event)
    backgroundImage:removeEventListener("touch",
onBackgroundTouch)
end
-- "create" is called whenever the scene is FIRST called
scene:addEventListener( "create", scene )

-- "show" event is dispatched whenever scene transition has
finished
scene:addEventListener( "show", scene )

-- "hide" event is dispatched before next scene's transition
begins
scene:addEventListener( "hide", scene )

return scene

```

A few notes on troubleshooting when using external files.

Often when you begin to try to find your error when working with external files, you will see the first file listed as causing the first error (i.e. main.lua) and it will list the line number that makes the call to the next scene. Ignore this error and continue reading through the

error log. The compiler is actually listing the stack of calls until it gets to the root error. So when you see “main:14” as the error source, and main.lua line 14 is calling the next lua file, ignore it and continue to the next line. Dig deep enough and you will find the source of all the errors.

There are a few commands that I didn’t use that would have provided better memory management if this were a much larger program. Using the various purge commands of the API (listed below), you can remove the previous scene from memory.

The composer API includes:

- composer.getScene() – returns the scene object. Can be used to call a function in a specific scene.
- composer.getSceneName() – returns the current scene name
- composer.getVariable() – returns the value of a variable set with `composer.setVariable()`.
- composer.gotoScene() – for transitioning to a new scene. Causes `exitScene` to be called.
- composer.hideOverlay() – hides the current overlay scene
- composer.isDebugEnabled - outputs debugging information to the Corona Terminal
- composer.loadScene() – used to preload a specified scene without causing a scene transition
- composer.newScene() – creates a new scene object
- composer.recycleOnLowMemory – If there a low memory warning is issued by the operating system, composer will recycle the least recent scene. To disable this feature, set the property to false.
- composer.recycleOnSceneChange – automatically removes previous scene from memory after the scene change. Default is false.
- composer.removeHidden() – removes all scenes except the current scene
- composer.removeScene() – purges the specified and unloads the scenes module
- composer.setVariable() – sets a variable that can be passed to subsequent scenes
- composer.showOverlay() – loads a scene above the current scene, leaving the current scene in-tact.
- composer.stage – returns a reference to the top-level composer display group.

### Project 6.3: 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 composer 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, I have created 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: Splash Screen

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, nil, 20)

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 (our background image) 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 be called when the user gives a ‘shout out’ to Dad and a second function to handle when the background is tapped. I chose to fade the splashGroup out over three seconds before calling the main function.

```

function hiDad()
    textObj = display.newText("Can I have $10?", 50,
display.contentHeight/2, native.systemFont, 24)
    textObj:setFillColor(1, 1, 1)
end

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. 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()
    hiDad()
end

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

Of course we could also use the composer API for creating a good splash screen (and we will in later projects), but it is always a good idea to see multiple methods for completing the same task.

## Summary

This chapter included a number of essential elements for making a good user experience in your app. We have examined the user experience, how to load custom fonts, how to use groups, the composer API for creating different views, and how to add a splash screen.

## Questions

1. What is a group and why are they useful?
2. True or False: A custom font does not have to be included in the app folder.

## **Assignments**

1. Create a project using 2 different fonts.
2. Create a splash screen for one of your previous projects that uses a display group to show and then hide the splash screen.
3. Using Project 6.2 as a starting point, add a splash screen using the composer API.
4. Using Project 6.2 as a starting point, add a splash screen using the composer API and use 2 different custom fonts for the mom.lua and dad.lua scenes.

# Chapter 7 Working with Media

## Learning Objectives

Sound, music, photos, and movies make up a very important part of the mobile device culture. In this chapter we are going to look at using, playing, and recording all four of these types of media. Specifically, we will learn to:

- the difference between loading and streaming audio
- load and play sound effects
- load and play long sound files such as music files
- use the built in camera on a device
- play movies in your app

## Audio

Sound effects and audio 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. Corona currently supports up to 32 distinct audio channels.

## Sound File Types

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.

For best performance with .mp3, use mono instead of stereo. It makes the file size significantly smaller and will improve your apps performance.

## 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. In my experience I have found that preloading sound effects makes the sound system very responsive. Since streaming sound must be decompressed at the time of play, it is not quite as responsive if you have a lot happening within the app.

## 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 and pre-processes the entire sound file into memory. The sound file can then 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 blow-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. The nice thing about using the audio system in this fashion is that Corona will manage your audio channels for you, releasing previously used channels for future use when the current sound effect is finished.

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.

## 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=l] [/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 – time 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.

### 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.

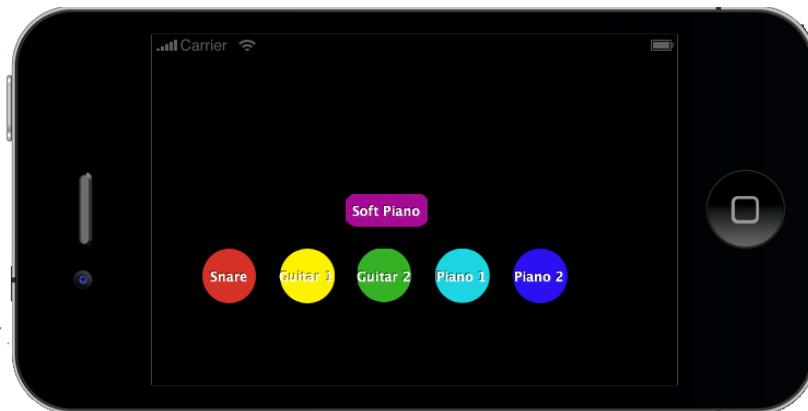
- `audio.findFreeChannel( [startChannel] )` - returns the channel number of an available channel or 0 if no channels are available.
- `audio.freeChannels` – returns the number of channels that are available.
- `audio.isChannelActive( channel )` – returns true if specified channel is playing or paused.
- `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.

## Project 7.0: 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 a new command: system.activate("multitouch"). system.activate("multitouch") is a required command for any app that will be accepting multiple, simultaneous touches.

### 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.

-----
-- 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("softpianosounddcab.mp3")
```

After setting our system for multi-touch, 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. 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 = display.newImage( "Button1.png", w,
display.contentHeight-100)

local guitar1Button = display.newImage( "Button2.png", w*2,
display.contentHeight-100)

local guitar2Button = display.newImage( "Button3.png", w*3,
display.contentHeight-100)

```

```

local piano1Button = display.newImage( "Button4.png", w*4,
display.contentHeight-100)

local piano2Button = display.newImage( "Button5.png", w*5,
display.contentHeight-100)

local mp3Button = display.newImage( "Button6.png",
display.contentWidth/2 -25, display.contentHeight/2)

snareButton:addEventListener("tap", playButton1)
guitar1Button:addEventListener("tap", playButton2)
guitar2Button:addEventListener("tap", playButton3)
piano1Button:addEventListener("tap", playButton4)
piano2Button:addEventListener("tap", playButton5)
mp3Button:addEventListener("tap", playButton6)

```

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.

### **Where did I put that file?**

As we prepare to discuss recording and accessing external files such as media files and photos, let us take a moment to discuss the directories on your mobile device. Both iOS and Android create a ‘sandbox’ around your app so that it is unable to impact other apps that are running on the mobile device. Each app has access to three folders: the Resource Directory, Documents Directory and Temporary directory. To access any of these folders you must place the keyword system in front of it. Let us look at each of these types of folders:

- 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. The Resource Directory is assumed (i.e., the default folder) when loading assets for your app, so you don’t have to specify it when loading an image or sound file.
- system.DocumentsDirectory – should be used for files that need to persist between sessions. When used in the simulator, the user’s documents folder is used.
- system.TemporaryDirectory – Just as the name says, is temporary. Only use for in app, temporary data. No guarantee that the file will be there the next time the app is used.

When you are programming and need to know where a file is stored on the phone, you can use the command: system.pathForFile(*filename [, baseDirectory]*) to find the absolute path to access files. This command will return nil if file does not exist.

## Movies

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 notify your app when the video has ended (i.e. give it a function to be called when the user is done playing the video). iOS supported formats include .mov, .mp4, .m4v, and .3gp using H264 compression at 640x480 at 30fps and MPEG-4 Part 2 video.

At the time I am writing this, playback is not yet supported on the Windows version of the Corona Simulator.

*media.playVideo(path [,baseSource], showControls, listener )* – plays the video in a device-specific popup video media player.

Playing a movie might look something like this:

```
local function doneWithVideo()
    print ("Video finished playing")
end

media.playVideo("catplayspiano.mov", true, doneWithVideo)
```

## 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. Since there is not a real camera in the simulator, it will instead open a finder window to allow you to select an image file to substitute for the camera image.

*media.show(imageSource, listener) –. imageSource can be: media.PhotoLibrary, media.Camera, media.SavedPhotosAlbum.*

## Project 7.1 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/>

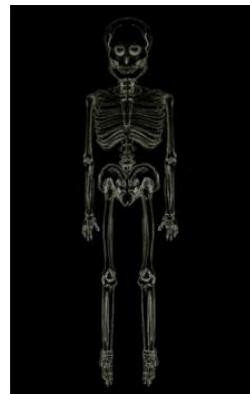


Image: Christina Cheek

Before we dive into the code, a reminder: The camera API is not available through the simulator. You will be able to select an existing 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 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 proceedButton
display.setStatusBar(display.HiddenStatusBar)

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( .5, 0, 0 )

local text = display.newText( "Tap anywhere to launch Camera", 0, 0,
nil, 16 )
text:setFillColor( 1, 1, 1 )
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= display.newImage("button.png")
            proceedButton.x = 160
            proceedButton.y = 340
            proceedButton:addEventListener("tap", processing)
            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 a later chapter.

## Recording Audio

It is possible to record audio using the Corona interface. While different platforms support different formats, both Apple and Android support the raw audio file format. The recording commands are:

- `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).

## Summary

As you can see, the media capabilities of Corona are quite extensive. 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.

## 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.
- 6) Create an app that makes a sound or plays music after a specific length of time. Great for people like me who lose track of time when they are doing something they enjoy!

# Chapter 8: A Little Phun with Physics

## Learning Objectives

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 Corona Labs 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 asleep, 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.

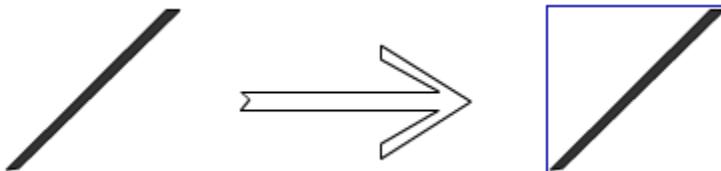
**Density** is multiplied by the area of the body's shape to determine its mass. The basis of this calculation is that 1.0 is equivalent to water. If a material has less mass than water, such as wood (or a duck or very small rocks - plus 5 points for those who get the reference), the density should be 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 body shape rectangle can simplify calculations, but 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 or the new `graphics.newOutline()` command. 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})
```

You can also set the body to be a circle instead of a rectangle, you can set the radius of the object to better handle collisions:

```
physics.addBody( object, [bodyType,] { density=d, friction=f,
bounce=b, radius=r })
```

## 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 damped. Default is zero.
- body.angularDamping – Number. Determines how much a body's rotation should be damped. 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.

## 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 8.0: 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 initial value for gravity
physics.setGravity(0, -0.1)

-- initialize gx and gy to store gravity changes
gx = 0
gy = -0.1

-- A couple of variables to make locating the sides easier
local xCenter = display.contentCenterX
```

```

local yCenter = 768/2

-- create border area so object doesn't fall off screen
local ground = display.newRect(xCenter, 768, 768, 10)
ground:setFillColor(1,1,1,1)

local leftSide = display.newRect(5,yCenter, 10, 768)
leftSide:setFillColor(1,1,1,1)

local rightSide = display.newRect(763,yCenter,10,768)
rightSide:setFillColor(1,1,1,1)

local top= display.newRect(xCenter,0,768,10)
top:setFillColor(1,1,1,1)

-- 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 pre-collision 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 pre-collision 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 pre-collision, post-collision can be a noisy event generator. It is best to keep the event local and screen out small post-collision 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, whereas a normal body-body collision will cause 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 include:

- 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, ratio )
```

Read only properties of the pulley joint include:

- .length1() – returns the distance between the 1<sup>st</sup> joint and the stationary pulley anchor point.
- .length2() – returns the distance between the 2<sup>nd</sup> 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.

Now that we have seen the API for physics, let's put some of these into action.

## Project 8.1 Sample Physics Projects

One of the most confusing projects for many people getting started with physics programming is how to use the various joints. To help solve that problem, I am providing a couple of short programs that will demonstrate some of the joints and how it can be used.

### 8.1A: Touch Joint

In my opinion, one of the most misunderstood joints is the touch joint. The touch joint creates a link between the user's touch and the object that they are touching, allowing the user to drag the object around the screen, yet the object still interacting with the physics environment with collisions. The touch joint creates an elastic connection between the touch and the object touch. The object will attempt to follow the user's touch, but will still be impacted by gravity or other on-screen physics objects.

main.lua

```
display.setStatusBar(display.HiddenStatusBar)
local physics = require("physics")
physics.start()
```

After hiding our status bar and starting physics, the first step is to create a function to handle the movement of the user's finger on the screen, which will drag the box around the screen.

The dragBody function is fairly standard and you will see it several more times as we work our way through more advanced projects. A dragBody function will generally have three (3) event phases that need to be handled: began, moved, and ended. These directly correspond with the start of the touch event, the user moving/dragging their finger on the screen, and the user releasing or lifting their finger from the screen.

After the creation of the function, we are going to create three (3) variables: body, phase, and stage. Body refers to what the user is going to drag (the crate or box in this case). Phase will store the current phase of the event: began, moved, or ended. The stage variable makes sure that the only thing we are moving is the box.

```
-- handling touch joint events
local function dragBody( event )
    local body = event.target
    local phase = event.phase
    local stage = display.getCurrentStage()
```

We are now ready to handle the 'began' stage. This is when the user has initially touched the crate and is ready to start moving it. In this stage we set the focus of the environment to the box and create the touch joint between the users touch and the place they tapped on the box. You could give the x and y location as the center of the object, but I thought it would be more interesting for you to see what happens when the user touches a corner of the box and starts to drag it around the screen.

```
if "began" == phase then
    stage:setFocus( body, event.id )
    body.isFocus = true

    -- Create a temporary touch joint and store it in the
    object for later reference
    body.tempJoint = physics.newJoint( "touch", body,
event.x, event.y )
```

Next, we will handle the 'moved' phase. The only thing that needs to be done in the move phase is to tell the box to try to follow where the touch is currently located (i.e. where the user moved their finger to). This is done with the setTarget method by passing the current location of the touch event.

```
elseif body.isFocus then
    if "moved" == phase then
        -- Update the joint to track the touch
        body.tempJoint:setTarget( event.x, event.y )
```

The final part of the dragBody function is to handle the release of the object. This occurs when the user stops touching the screen. First, the stage is returned to normal (i.e. the box is no longer the focus). After resetting the focus, we remove the touch joint so that it will no longer be active. Finally, we do a return true, to make sure that the app doesn't continue to try to continuously perform touch events while we are handling the initial event.

```
elseif "ended" == phase or "cancelled" == phase then
    stage:setFocus( body, nil )
```

```

        body.isFocus = false

        -- Remove the joint when the touch ends
        body.tempJoint:removeSelf()

    end
end
-- Stop further propagation of touch event
return true
end

```

Now the basic program: A little text to tell us what is happening, loading the box and ground, adding them to the physics engine and setting up the event listener for the touch joint event, which is initialized by a touch event (remember, a touch event is when the user continues to touch the screen, whereas a tap event is a quick touch and release).

```

local mytext = display.newText("Touch Joint",0 ,0, nil, 36)

local box = display.newImage("crate.png",
display.contentWidth/2, 0)
local ground = display.newImage("ground.png", 0,
display.contentHeight - 50)

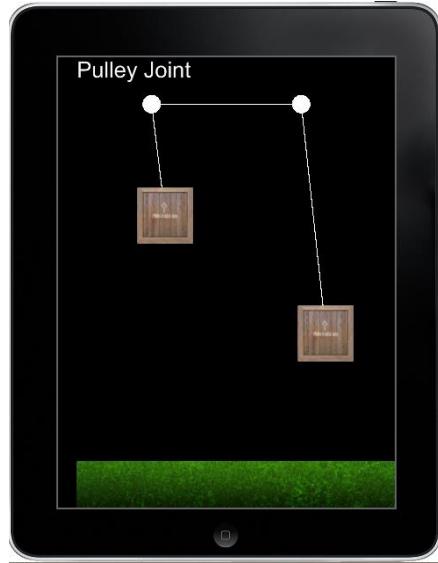
physics.addBody(ground, "static")
physics.addBody(box, "dynamic", {density= 1, friction =0.3,
bounce = 0.5 })

box:addEventListener("touch", dragBody)

```

### **8.1B: Pulley Joint**

Compared to the touch joint, the pulley joint is fairly straight forward, though it has a great deal of built-in power and flexibility. With the pulley joint you can simulate a two pulley system that can handle ratios, thus allowing you to simulate block and tackle configurations.



For our example, I am going to use both pulleys and show how it can be used to simulate simple pulley systems. As before, we start by hiding the status bar, enabling physics, displaying some text as to what the app will demonstrate, and loading two crates and the ground. Then two circles are created to represent the pulleys and a connecting line is drawn from the center of the boxes to the pulleys. We move the two lines to the back so that they are not in front of the boxes and add the boxes to the physics engine. I set box2 to be slightly denser than box1 so that we can see the pulleys in action.

#### main.lua

```
display.setStatusBar(display.HiddenStatusBar)
local physics = require("physics")
physics.start()

local mytext = display.newText("Pulley Joint",0 ,0, nil, 24)

local box1 = display.newImage("crate.png",
display.contentWidth*0.25, 200)
local box2 = display.newImage("crate.png",
display.contentWidth*0.75, 200)
local ground = display.newImage("ground.png", 0,
display.contentHeight-50)

local pulley1 = display.newCircle(display.contentWidth*0.25, 50,
10)
local pulley2 = display.newCircle(display.contentWidth*0.75, 50,
10)
line1 = display.newLine(box1.x, box1.y, pulley1.x, pulley1.y)
```

```

line2 = display.newLine(box2.x, box2.y, pulley2.x, pulley2.y)
line3 = display.newLine(pulley1.x, pulley1.y, pulley2.x,
pulley2.y)

line1:toBack()
line2:toBack()

physics.addBody(ground, "static")
physics.addBody(box1, "dynamic", {density= 1, friction =0.3,
bounce = 0.5 })
physics.addBody(box2, "dynamic", {density=1.01, friction = 0.3,
bounce = 0.5 })

```

To create the pulley, we just need to specify the two objects that will be on each end of the pulley (box1 & box 2), the pulleys x & y location, the two boxes starting location, and the ratio between the boxes. The default ratio is 1 or equal.

```

-- physics.newJoint("pulley", bodyA, bodyB, anchorA_x,
anchorA_y, anchorB_x, anchorB_y, bodyA.x, bodyA.y, bodyB.x,
bodyB.y, 1.0 )
local myJoint = physics.newJoint("pulley", box1, box2,
pulley1.x, pulley1.y, pulley2.x, pulley2.y, box1.x, box1.y,
box2.x, box2.y, 1)

```

To make the app more visually appealing, I update the lines every time the frame updates. Typically most apps run at 30 frames per second. Thus the lines are being re-drawn each time the app starts a new frame.

The RunTime:addEventListener ("enterFrame", listener) is a handy command to remember for your graphic intensive apps; rather than being time sensitive, the graphics can be updated each time the screen is updated.

```

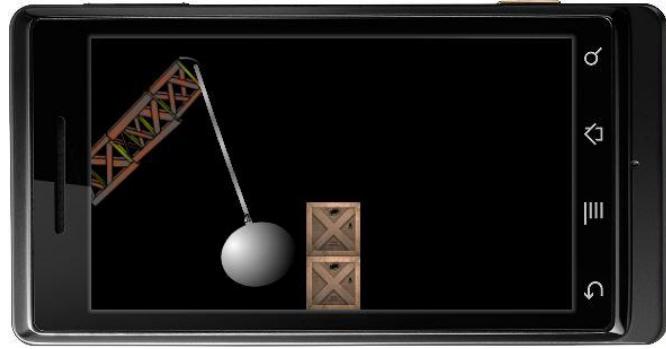
local function lineUpdate()
    line1:removeSelf()
    line2:removeSelf()
    line1 = display.newLine(pulley1.x, pulley1.y, box1.x,
box1.y)
    line2 = display.newLine(box2.x, box2.y, pulley2.x,
pulley2.y)
    line1:toBack()
    line2:toBack()
end

```

```
Runtime:addEventListener("enterFrame", lineUpdate)
```

## 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(1,1,1,1)
```

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 physics 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, and 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 8.0: 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 8.0. 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 Mobile Game Design

## Learning Objectives

One of the most exciting areas of mobile devices is being able to develop and play games. The concepts behind mobile game development are appropriate for an entire book or series of books. Consider this chapter a very brief introduction to some of the basic programming considerations. In this chapter we will learn:

- How to use timers
- Handling drag events
- Using frame base animation
- How to make a game

## Timers

Timers are one of the key components of game development. Typically within a game you will need to perform an action every so many seconds or milliseconds. By using the timer API you can call a function to control the functions of your game or app. The primary use of the timer API is `timer.performWithDelay`.

**`timer.performWithDelay(delay, listener [, iterations])`**- provides listener after a period of delay. The listener call cannot contain parameters in the actual function call. Passing parameters within the function call (the *listener*) of a `timer.performWithDelay` causes the function to be called immediately, thus rendering the delay ineffective. The *delay* is in milliseconds. *Iterations* is 1 by default; passing a 0 will create an infinite loop (until canceled).

Example:

`timer.performWithDelay( 400, gameLoop, 0)` -- will call the function *gameLoop* every 400 milliseconds until canceled.

Timer includes the additional methods of:

- `timer.cancel(timerId)` –cancels the timer operation.
- `timer.pause(timerId)` – pauses the timer operation.
- `timer.resume(timerId)` – resumes a paused timer operation.

## More on Touch/Multi-touch

Typically in most games you will have multiple touch events occurring at the same time. To handle those events, you will need to use multi-touch. To respond to a touch or multi-touch event, an event listener (i.e. a function) must be registered for the object (i.e. button) that the user is touching. Touch events have several phases (which can be found in the `event.phase` parameter that is passed to the event listener). The phases include:

- **began** – occurs when the user first touches the object
- **moved** - occurs if the user moves their finger as they are touching the object
- **ended & cancelled** – occurs when the user lifts their finger or the touch is otherwise canceled.

While it is not necessary to check for the event phase in many event listener situations, it can be very useful in many applications. For example, can the user drag items around on the screen in your game? Can they hold (i.e. mash) a button for continuous running or jumping? If so, then you will need to setup a multi-touch event to handle those situations.

Multi-touch can be activated with the command:

**system.activate("multitouch ")**

When multi-touch is enabled, multiple touches to the same object are handled as a single touch event. We will use multi-touch in our game example later in this chapter for moving the spaceship.

## setFocus

`setFocus` is used to make a display object the target for all future touch events. Generally `setFocus` is used for objects that change appearance or location after they are touched. It can be used in conjunction with multi-touch. To restore normal focus, pass nil in place of the obj. An example of using `setFocus` would be dragging an object on screen. We will use `setFocus` later in the chapter for our spaceship movement.

**stageObject:setFocus(obj)**

## 'enterFrame' Animation

In chapter 3 we examined creating animation effects using the `transition.to` command. Now we are ready for a method of animation that works a little differently. If you are creating animations or need to call a function at the beginning of each frame (a normal situation in game development), the `enterFrame` event is just what you need. The frame interval of your application can be set in the `config.lua` file at either 30 or 60 frames per second.

For better battery performance on mobile devices, 30 frames per second is recommended. Remember, just because a device can support a faster frame rate, doesn't mean it should.

### Project 9.0: enterFrame Animation

To show how you can use an enterFrame event, I have modified the animation project from chapter 3. Using the same graphics as before, we will use the enterFrame event and a function to move the square to the center of the screen.

First, we will need a config.lua file:

```
application =
{
    content =
    {
        width = 640,
        height = 960,
        scale = "letterbox",
        fps = 30,
        antialias = false,
        xalign = "center",
        yalign = "center"
    }
}
```

I used a higher resolution than typical for most games (the standard is 320 x 480 for older phones) so that we can better see the movement of the square. Note that the fps (frames per second) is set to 30.

main.lua

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

center.x = display.contentWidth/2
center.y= display.contentHeight/2

square.x = math.random( display.contentWidth )
square.y = math.random( display.contentHeight )

local function move()
    if (square.x > center.x) then
        square.x = square.x -1
    elseif (square.x < center.x) then
        square.x = square.x +1
    end
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

Runtime:addEventListener("enterFrame", move)

```

There are only three modifications from our original program in chapter 3:

- The movement of the square is now a function called move
- Removed the while...Do loop from the function
- Added the event listener for enterFrame, which calls the move function

When you try this app, you will find that the square will slowly move toward the center of the screen. It is moving at the rate of 30 pixels per second: 1 pixel per frame.

Note: The fps in config.lua has only two possible settings: 30 and 60. 30 is the default.

## Game Development

The field of game development, particularly mobile game development, is one of the fastest growing industries in the world. One of the reasons I first became interested in Corona was in large part due to how easy you could make an app or game with the same SDK. It is always nice to be able to leverage what you have learned in making business or personal apps for the creation of games.

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.



## Design Inspiration

Star Explorer is inspired partially from my misspent youth, spending all the money that I earned on 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.
- Starship must shoot the moving asteroids.
- Move the ship by dragging it.
- Fire by tapping on the ship.
- If an asteroid hits the ship, one life is lost.

We will start with 3 lives and keep track of the score for the player as well. I am beginning this project targeting it for the iPhone.

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



To keep everything simple for this game, we will use just one programming file (main.lua). To begin with, we will hide the status bar, enable multi-touch, 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 affecting to sleep in order to reduce the overhead on the processor.

### main.lua

```
-- Hide status bar
display.setStatusBar(display.HiddenStatusBar)
system.activate("multitouch")

-- 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 leads 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. Don't get too hung-up on arrays at this point; we will discuss them in greater detail in chapter 10. Think of them as being able to store multiple values in one variable.
- 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 initializing 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 = 400 -- time between game loops in milliseconds
local died=false
local explosion = audio.loadSound("sounds/explosion.wav")
local fire = audio.loadSound("sounds/fire.wav")

```

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 can be shown when we are ready to display them.

```

-- 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:setFillColor(1, 1, 1)
    textScore:setFillColor(1, 1, 1)
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 moving our starship. 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. 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. The touch event has 3 phases:

began, moved, and ended. When the ship movement first starts from a touch event, it becomes the focus (`t.isFocus`) for future ship-touch actions.

```
-- 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 initial 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, bounce to the default values. For this version of the game these values do not matter, but I might use them in a future version. The final line of code in this section, we give the starfighter a `myName` value. The `myName` is used later in collision detection so that we know that the starfighter was involved in the collision and

can respond appropriately. The last line of this function will keep the ship from rotating after it gets hit by an asteroid. This will ensure the ship is always pointing up or forward.



```
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, bounce=1.0})
    starfighter.myName="starfighter"
    starfighter.isFixedRotation = true
end
```

Time to load the asteroid body. Each asteroid is set with a low density (1.0), no friction (we are in space after all), 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 could not appear from the bottom of the screen; only from the sides or top.

We will need to keep track of how many asteroids are 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, friction
=0.4, bounce=1})
    local whereFrom = math.random(3)
    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. 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, 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  
        audio.play(explosion)  
        event.object1:removeSelf()  
        event.object2:removeSelf()  
        lives=lives -1  
        local lose = display.newText("You Have  
Failed.", 30, 150, nil, 36)  
        lose:setFillColor(1, 1, 1)  
    else  
        audio.play(explosion)  
        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 a 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  
    audio.play (explosion)  
    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})
audio.play(fire)
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 400 milliseconds 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
    end
end

```

```

        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 400ms (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 for it to be ready for the store. We have discussed how to do animation with enterFrame events, using multi-touch, and working with timers. In our next chapter we will delve more deeply into using arrays (also referred to as tables in Lua).

## 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: Tables and Arrays

## Learning Objectives

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 store large quantities of data.

In our examination of tables we will:

- Clarify the term table and array
- Examine the tools available for tables
- Create a simple table

## Tables vs. Tables vs. Arrays

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 (like a spreadsheet), 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 use the term table view to refer to the table view/list view associated with app development that has been used by Apple and can be reproduced widgets (Chapter 11). 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. The only time I will use the term “table” without qualification will be in reference to the table API, which we will discuss later in the chapter.

I should note that Corona does have a table command and a table widget in the API. The table command refers to the Lua array table. This should not be confused with the table widget (which we will discuss in Chapter 11) that emulates a table or list view.

Confused? I was initially as well! For now, we are just going to focus on Lua array tables and the table API. We will save table or list views for Chapter 11.

## Introducing Arrays

When teaching programming, one of the dividing lines between the novice programmer and the intermediate programmer is the understanding of the concept of arrays. If you can get this concept, you will be set for a whole new world of programming concepts and items that you can create.

I have always found it easiest to conceptualize an array by picturing a single column in a spreadsheet. If I wanted to create an array of the first names of students in my class, it might look something like:

```
myStudents =
```

Joe
Jean
Fred
Cindy
Mary

In this example, I have 5 students. All of the students are stored in one variable: myStudents. Because they are all stored as one variable, I am able to work with them easily as a group of records.

Each part or row of the array is referred to as an element. Thus “Joe” is the value of the first element of the array.

To create an array in Lua, you use curly brackets in the variable declaration:

```
local myStudents = {}
```

You can also declare the contents of the array:

```
local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}
```

**Note:** If you want Corona to treat the names as strings, they have to be in quotations; else they will be treated like variables.

Now I can easily access each of the students using a for...do loop:

```
local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}  
for count1 = 1, 5 do  
    print (myStudents[count1])  
end
```

Terminal Output:

```
Joe  
Jean  
Fred  
Cindy  
Mary
```

## Table API

We have several very useful commands that can be used through the table API:

- `table.concat(array [, string, number1, number2])` – concatenates the elements of an array to form a string. Optionally, you can pass a string to be inserted between the values (such as “,”). Number1 and number2 refer to the index of the elements to be concatenated. Number1 must be less than number2. If omitted, number1 will be the first element in the table and number2 will be the last element.
- `table.copy(array)` – creates a copy of a table. Multiple tables can be included in the copy.
- `table.indexOf(array, element)` – returns the index number of the element of a table. In other words, it will search the array for the supplied element and return the index number of that element.
- `table.insert(array, [position,] value)` - inserts the provided value into a table. If a position is supplied, the value is inserted before the element currently in that position.
- `table.maxn(array)` – returns the largest positive index number of a table (i.e. the last positive index number).
- `table.remove(array [, position])` – removes the table element in the supplied position. If a position is not provided, then the last element in the array is removed.
- `table.sort(array [, comparison])` – sorts the table element into a given order, updating the table to the new sorted order. By default < (less than or alphabetical order) is used. If a comparison is supplied, it must be a function that receives two table elements.

Let's examine the usage of each of these API commands:

### Concatenation:

When concatenating an array, you are creating a string using the elements within the array:

```
local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}  
print (table.concat(myStudents, ",") )  
  
--Output:  
Joe, Jean, Fred, Cindy, Mary
```

### Copy:

Copy returns a duplicate of the table or tables passed to it. It can be used to join additional arrays:

```

local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}
local myStudentsScores = ( 98, 87, 68, 100, 89)
local newArray = table.copy(myStudents)
print(table.concat(newArray, ", "))

-- Output:
Joe, Jean, Fred, Cindy, Mary

local newArray2 = table.copy(myStudents, myStudentsScores)
print (table.concat(newArray2, ", "))

--Output:
Joe, Jean, Fred, Cindy, Mary, 98, 87, 68, 100, 89

```

### **indexOf:**

The indexOf API command returns the index of a supplied element that is in the array. If the element is not present in the array, nil will be returned.

```

local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}
print (myStudents, "Jean")

-- Output:
2

```

### **insert:**

Inserts a new element into an array. If a position is not provided as an argument, the new element will be added as the last element of the array. If a position is provided, the new element will be inserted before the element previously at that index (i.e. what was the second element becomes the third element).

```

local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}
table.insert(myStudents, 2, "Jeff")
print(table.concat(myStudents, ", "))

-- Output:
Joe, Jeff, Jean, Fred, Cindy, Mary

```

### **Maxn:**

By using the table.maxn() API, it isn't necessary that I know how many elements are in the array:

```
local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}
```

```

for count1 = 1, table.maxn(myStudents) do
    print (myStudents[count1])
end

Output:
Joe
Jean
Fred
Cindy
Mary

```

In this example, `table.maxn(myStudents)` returns the number of elements in the array, making it much easier to work with tables of an unknown size. This can only be used in cases where the index is numerical. A non-numerical index will return nil causing an error.

Note: In some languages, the index of the first element in an array is element 0. In Lua, the index of the first element is 1.

### **Remove:**

The remove API deletes from the array the element at the provided index position. If no index position is provided, the last element will be deleted from the array.

```

local myStudents = {"Joe", "Jeff", "Jean", "Fred", "Cindy",
"Mary"}
table.remove(myStudents, 2)
print(table.concat(myStudents, ", "))

-- Output:
Joe, Jean, Fred, Cindy, Mary

```

### **Sort:**

The sort API command sorts the table into alphabetical or numerical order if an operand is not provided. If a different sort order is needed, you will need to supply a function that will return true or false if the sort condition is not met. For example, if you wanted to sort an array in reverse-alphabetical order (i.e. `>`), then you would need a code such as:

```

local myStudents = {"Joe", "Jean", "Fred", "Cindy", "Mary"}

local function compare(a, b)
    return a > b
end

```

```

table.sort(myStudents, compare)

print(table.concat(myStudents, ", "))
-- Output:
Mary, Joe, Jean, Fred, Cindy

```

Sort will pass the elements two at a time for our compare function to determine if the first element is greater than the second element. Thus, in the first instance, Joe and Jean will be passed. Since "Joe" is greater than "Jean" (at least alphabetically), the compare function will return the value TRUE, which tells the sort API it doesn't need to do anything. In the case of Joe and Mary, it will return FALSE, causing the sort API to place "Mary" before "Joe".

## Flexibility of Lua Array Tables

Lua Array tables are very flexible in the content that they can contain. They are commonly used to handle events and a return from a function. In most programming languages, arrays must be of a specific data type such as integer, string, floating decimal, Boolean, etc. In Lua we have more flexibility. Lua array tables can be heterogeneous. They can contain any type of data except nil.

Once you have declared a Lua array table, you can also use it to represent records or objects with field names like we did in the last chapter when we created our asteroids and shots fired. The process is simple. Once you have created your array, you can create your own field names as needed by placing a period after the variable name, just like you would access a property for the variable such as the x or y location:

```

local myArray = {}

myArray.id = 1
myArray.myName = "Array 1"
myArray.x = 10
myArray.y = 50

```

You can also use a Lua array table to describe an object. For instance, if I was creating an RPG (Role Playing Game), I might create a player object such as:

```

local player1 = {}
player1.location = level
player1.class = "fighter"
player1.name = "Conan"
player1.weapon = "sword"

```

```

player1.health = 10
player1.image = display.newImage("player1.png")

```

I think you get the idea. Obviously Lua array tables are very flexible and can help organize your app data in many useful ways. All of this information is now associated with the variable player1 and can be accessed through its properties. Note that you can associate images, sounds, or even functions to an element of an array table.

## The 4<sup>th</sup> dimension and beyond

That takes care of a single or one-dimensional array, but what about multi-dimensional arrays? Yes, it is possible (and often necessary) to create arrays that are 2, 3, 4, or more dimensions! To keep it simple, we are just going to look at 2-dimensional arrays. The same concepts apply if you find yourself needing to build more complex arrays.

To create multi-dimensional array tables, you can assign an array table within an array table. To simplify working with multi-dimensional tables, I recommend that you always use a numeric index:

### Project 10.1 Multi-dimensional Array

```

local myArray = {}
myArray[1]={"Joe", "Jean", "Fred", "Cindy"} -- first name
myArray[2] = {"Smith", "Smith", "Smith", "Smith"} - last name
myArray[3]={"142 Main", "163 South St."} -- address
myArray[4]={} -- city
myArray[5] = {} -- state
myArray[6] = {} - zip code

for i = 1, 6 do
    for j = 1,4 do
        print (myArray[i][j])
    end
end

```

This creates a 2-dimensional array:

```
myArray =
```

Joe	Smith	142 Main			
Jean	Smith	163 South St.			
Fred	Smith				
Cindy	Smith				

**Note:** How you choose to conceptualize the array is a matter of personal preference. In the first visual, I used a column to represent a 1-dimensional array. In this visual I have instead used the first dimension to represent a row, with the second dimension representing the columns.

Just like before, we have complete flexibility in what we can store within the Lua array table. One warning: be very careful when storing certain information in any type of variable. Numbers such as telephone numbers, social security, driver's license, etc. should be treated as strings and not as numbers. In other words, storing 321-555-4651 is very different than "321-555-4651". The first number will be treated as a number and calculated resulting in -4,885. The second, since it is enclosed in quotes will be treated as a string and retain its original meaning.

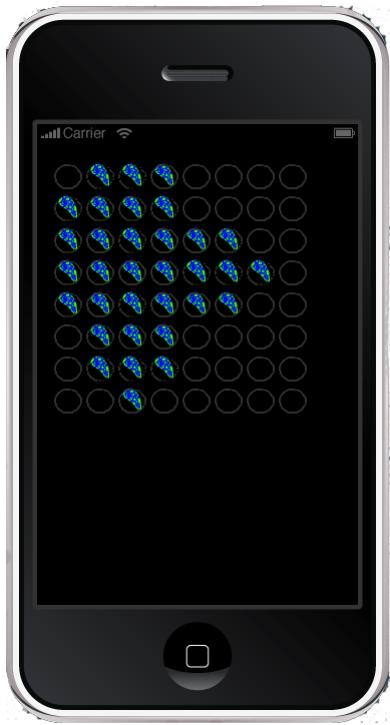
To cycle through a more complex array such as what we have in our example, you will need to use a nested loop:

```
for i = 1, 6 do
    for j = 1, 4 do
        print (myArray[i][j])
    end
end
```

To access each dimension of an array, use a bracket for the index number. For example: If I have the command `print( myArray[1][2] )` the result will be Jean; the 1<sup>st</sup> column, 2<sup>nd</sup> row. As you can see, I did not fully populate the array. When the app runs, the empty elements will be returned as nil.

Now that we have been exposed to Lua array tables, let's see it in action for a simple game.

## Project 10.2 Conway's Game of Life



John Conway's Game of Life is a classic example of cellular automation (as in cells in the body, not cell phones). Devised in 1970, it is a zero-player game; it shows evolution of life based upon a few simple rules. It requires no input from the player, but will do nicely to demonstrate how array tables can be used in a game.

First the rules (modified from Wikipedia:  
[http://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)):

The universe of the Game of Life is an infinite two-dimensional grid of square *cells*, each of which is in one of two possible states, *alive* or *dead*. Every cell interacts with its eight *neighbors*, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives onto the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the *seed* of the system. The first generation is created by applying the above rules to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a *tick* (in other words,

each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

The following code is based upon a sample code and has been modified to work in the Corona SDK environment.

To get started, we will setup our environment. The variable m is used to specify the number of rows and columns in the “universe.” The game will go through 10 iterations (num\_iterations). We will create the array tables for storing the life process. To make it more visually appealing, we will use graphics to represent the life process.



### Main.lua

```
-----
-- Conway's Game of Life in Corona
-- Chapter 10.2
-- Demonstration of multi-dimensional Arrays
-----

local m = 8          -- number rows / columns
local num_iterations = 10
local myCell = {}
local cell = {}
local iteration = 0
```

Next we will create the function setup. Setup will create our initial universe based upon m (which is currently set to 8, thus an 8 by 8 array). I am loading an image to all cells as part of the initialization to represent the base or no-life environment.

The images are both 32x32 pixel, so placement can be easily accomplished with just a bit of math. We'll start our 'life' at cells on the 3<sup>rd</sup> row in columns 2 through 6.

```
local function setup()
    for i = 1, m do
        cell[i] = {}
        myCell[i] = {}
        for j = 1, m do
            cell[i][j] = 0
            myCell[i][j] = display.newImage("base.png")
            myCell[i][j].x = (i * 32)
            myCell[i][j].y = (j * 32) + 20
```

```

        end
    end

cell[3][2] = 1
cell[3][3] = 1
cell[3][4] = 1
cell[3][5] = 1
cell[3][6] = 1

end

```

The Evolve function is the heart of the program. It receives the cell array and steps through it with a for...do loop, first making a copy (cell2) then checking to see if life continues, grows, or dies based upon the count of surrounding life. After all the calculations are completed, the cell array is returned with its new values.

Notice that life is represented as a value of 1 and no life is represented by 0 through the entire process.

```

local function Evolve( cell )
    local m = #cell
    local cell2 = {}
    for i = 1, m do
        cell2[i] = {}
        for j = 1, m do
            cell2[i][j] = cell[i][j]
        end
    end

    for i = 1, m do
        for j = 1, m do
            local count
            if cell2[i][j] == 0 then count = 0 else count = -1
        end
        for x = -1, 1 do
            for y = -1, 1 do
                if i+x >= 1 and i+x <= m and j+y >= 1 and
j+y <= m and cell2[i+x][j+y] == 1 then count = count + 1 end
            end
            if count < 2 or count > 3 then cell[i][j] = 0 end
            if count == 3 then cell[i][j] = 1 end
        end
    end

    return cell
end

```

The updates function is called every one and a half seconds to update the life universe. It is a simple for...do routine to display either life or no life on the board. Once the board has been updated, the cell array is updated by passing cell to the Evolve function above.

```
local function updates()
    for i = 1, m do
        for j = 1, m do
            if cell[i][j] == 1 then
                myCell[i][j] = display.newImage("life.png")
                myCell[i][j].x =(i * 32)
                myCell[i][j].y= (j * 32)+20
            else
                myCell[i][j] = display.newImage("base.png")
                myCell[i][j].x =(i * 32)
                myCell[i][j].y= (j * 32)+20
            end
        end
    end
    cell = Evolve( cell )
    iteration = iteration + 1
    print(iteration)

end
```

Now we can launch the program by calling the initial setup function and then setting a timer.performWithDelay to update the screen every 1 ½ seconds for the number of iterations called for at the beginning of the program.

```
setup()

timer.performWithDelay(1500, updates, num_iterations)
```

Hopefully seeing Conway's Game of Life will trigger some of your own ideas on game or app development.

## Summary

You now have a taste of working with arrays. Congratulations! This concept is what is generally considered to separate the beginner from the intermediate programmer! It will take some time to become comfortable working with arrays, but you are now well on your way! Now that we have the fundamentals of programming and app development, we can jump into some of the interesting tools available for creating interesting apps on smartphones and tablets in our next chapter.

## Assignments

1. Create a simple array showing exam scores for a class of 6 students: 85, 67, 92, 42, 99, 77. Using table.insert, add an additional score of 96.
2. Building on assignment 1, sort the array in ascending order.
3. Building on assignment 1, remove the 3<sup>rd</sup> element.
4. Create a lamp object that has the following properties:  
Color: blue status: off      bulb: 60W      power: battery  
Then write a function to change the lamp status to on if it is off, or off if it is on.
5. Create a two-dimensional array that shows 5 student's names in the first column and their current class grade (in percent) the second column.
6. Challenge: Building on assignment 5, add the following features:  
Search for student by name and change their grade.  
Add a new student  
Remove a current student.

# Chapter 11: Going Native - Working with Widgets

## Learning Objectives

Time to jump into tools that will make your apps user interface appear native. These tools or widgets allow us to add picker wheels, buttons, scroll-bars and sliders with ease. We will also examine tools for creating mock-ups. Specifically we will learn:

- Examine various mock-up or pre-design tools
- How to 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

## Mock-ups and pre-design tools

Think back to one of the classic Disney films you saw as a child. Whether it was Snow White, Sleeping Beauty, or one of the many other Disney classics, you were watching a film that had been carefully planned and designed. Walt Disney was known for his long storyboarding sessions to carefully communicate the flow and design of the entire movie. This attention to detail is what separates the professional from the amateur. When designing apps or games, while it is sometimes still called storyboarding, the process of pre-planning your project is also called a mock-up, flow, or conceptualization of the project.

On one project that I was working on several years ago, we were to develop an app for the iPad. The problem was, the iPad wasn't even released. Yes, that's right, we were developing an app for a product that hadn't even been released. To help all of us, the artists, clients, and programmers, have a better concept of what we were developing, one of the team leaders created 5 wooden versions of the "iPad" so that everyone on the project could get a feel for the device. This proved to be a very useful mockup, and the wooden mockups were much prized and sought after at the end of the project.

Whether working by yourself or on a team, being able to conceptualize the project and try different configurations will save you a great deal of time. When working on a team or for a client, the ability to show the concept and possible screen flow or arrangement objects in the app saves misunderstandings and can greatly improve.

When creating such projects, it is critical to show the screen-flow as different parts of the app are used. Showing what screen will become active is critical for understanding the complexity and flow of the project. It can also help you to avoid dead ends or overly complicated designs.

While creating a mock-up or pre-design of an app can be done with a pencil and paper, many developers prefer to use tools that speed up the design process even more. In Chapter 20 I have included several mockup tools that are popular with mobile developers. The majority of these tools take advantage of the pre-built widgets that are available in iOS and Android devices. So let's talk about widgets!

## Widgets

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.

Be aware that widgets have gone several stages in Corona. We are now on Widgets 2.0 (as of this writing), which are much more sophisticated and consistent in their usage compared to the first version.

To use a widget in your app, you must load the widget library prior to using any objects:

```
local widget = require( "widget" )
```

The appearance of the widgets is controlled by image sheet objects. You can find sample image sheets from which you can customize (or just used in your apps) in the CoronaSDK > SampleCode > Interface > WidgetDemo > assets folder.

Some of the widgets that you can use in your app include:

- **Buttons** - the `widget.newButton` provides a button that supports `onPress`, `onRelease`, and `onEvent` events.
- **Picker Wheel** – The picker wheel widget allows the user to rotate a dial to select their response for the application using tables to store each column of data.
- **Progress Bar** – The progress bar can be used show startup, download, upload or any number of situations where a progress bar is needed.
- **ScrollView** - The scroll view widget allows you to create scrolling content areas. 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.
- **Segmented Control** – The segmented control is allows you to setup multiple buttons where only one can be selected at a time.
- **Slider** - The `widget.newSlider` allows you to create a slider object that can be adjusted in width.

- **Spinner** – The spinner widget is a useful tool for showing background processing or loading so that users know that the app is functioning and to be patient.
- **Stepper** – The stepper widget is used to increase or decrease (increment or decrement) a value. You have control as to how much the value is changed.
- **Switch** – The switch widget is a very flexible tool. It is used to show a binary situation (on/off, true/false, 1/0), but can do this as a radio button, checkbox, or on/off switch depending on your needs.
- **Tab Bar** – The tab bar allows you to create a customizable tab bar. Tabs are auto-positioned based upon the number of buttons.
- **Table View** - The table view widget allows you to create scrolling lists. With this widget you can control the rendering of the individual rows.

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)
```

### Making Your Widgets Look Good

If you would like to use a theme that either makes your widgets look like native Apple iOS or Android widgets, you can use the `widget.setTheme()` API command. This should be called immediately after the loading of the `widget` API (i.e. right after you have the command line `local widget = require ("widget")`).

The theme files are Lua files with tables that correspond to each of the widgets. The easiest way to get started creating your own custom theme is to edit the existing widget themes. You can find a sample of the theme files in the `widget demo` sample file under the Corona Folder.

To set a specific theme for the widgets that are either iOS or Android themed, you simply include the line:

```
widget.setTheme("widget_theme_ios")
or
widget.setTheme("widget_theme_android")
```

## **widget.newButton**

The first widget will be a popular addition to your apps. The button widget adds some nice features that make it easy to use within your apps instead of having to create your own button each time. There will be times when you will want to use your own customized button, but for most instances, being able to use the widget button will be your first choice.

The `widget.newButton` provides a button that supports `onPress`, `onRelease`, and `onEvent`. The button widget is very powerful and flexible with many options.

### **Button Parameters:**

- `id` – an optional string that can be used to identify the button (default is “`widget_button`”).
- `left, top` – initial coordinates of buttons left, top corner (default is 0, 0).
- `width, height` – allows adjusting the buttons width and height. Should be set to the size of your button image or if using the 9-piece button, it will scale.
- `label` – text that will appear on the button.
- `labelAlign` – text alignment of button label. Valid options are “left”, “right”, and “center” (default is “center”).
- `labelColor` – RGBA (red, green, blue, alpha) table showing default and over color states of the label text.
- `labelXOffset, labelYOffset` – adjust the x & 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` - will allow the text to appear embossed if set to true (default is true).
- `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.
- `onEvent` – optional function and should only be used if none of the other above events are used. The callback function will need to test for `event.phase`.
- `isEnabled` – optional. If false, the button will not respond to touch events (default is true).

### **Methods**

- `setLabel(string)` – Changes the button’s label text.
- `getLabel()` – returns the buttons current label text.
- `setEnabled()` – set to false to disable the button.

The button image can be implemented in one of three ways:

### **Button from an image file (i.e. loading a `button.png`):**

- defaultFile, overFile – 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).

**Button – 2 frame image sheet:**

- sheet – the image sheet containing the images for your button.
- defaultFrame – frame number containing the default image of the button.
- overFrame – frame number containing the pressed (over) image of the button.

**Button – 9 slice image sheet:**

- sheet – the image sheet containing the images for your button.
- topLeftFrame, topLeftFrameOver – default and pressed frame numbers for the top left slice of the button.
- middleLeftFrame, middleLeftFrameOver – default and pressed frame numbers for the middle left slice of the button.
- bottomLeftFrame, bottomLeftFrameOver - default and pressed frame numbers for the bottom left slice of the button.
- topRightFrame, topRightFrameOver - default and pressed frame numbers for the top right slice of the button.
- middleRightFrame, middleRightFrameOver - default and pressed frame numbers for the middle right slice of the button.
- bottomRightFrame, bottomRightFrameOver - default and pressed frame numbers for the bottom right slice of the button.
- topMiddleFrame, topMiddleFrameOver - default and pressed frame numbers for the top middle slice of the button.
- middleFrame, middleFrameOver - default and pressed frame numbers for the middle slice of the button.
- bottomMiddleFrame, bottomMiddleFrameOver - default and pressed frame numbers for the bottom middle slice of the button.



### Project 10.0 widget.newButton Example

```
local widget = require( "widget" )

-- Function to handle button events
local function handleButtonEvent( event )
    local phase = event.phase

    if "ended" == phase then
        print( "You pressed and released a button!" )
    end
end

-- Create the button
local myButton = widget.newButton
{
    left = 100,
    top = 200,
    width = 150,
    height = 50,
    defaultFile = "default.png",
    overFile = "over.png",
    id = "button_1",
    label = "Button",
    onEvent = handleButtonEvent,
}
```

## **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. The picker wheel only accepts one argument, which is an array table that can contain the following parameters:

### **Parameters:**

- id – string to identify the picker wheel (default is “widget\_PickerWheel”).
- left\_top – top, left corner of the widget.
- 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).
- fontColor – RGBA table for text color of each column (default is black).
- columnColor – RGBA table for column background color (default is white).
- columns – table array that will hold sub-tables arrays representing the individual columns of your picker wheel.

### **Methods:**

- getValues() – returns the selected values from the 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.
  - align – sets text to left, right, or center alignment (default is center).
  - labels – stored in a table, sets the label for each row.
  - Methods:S
- 
- picker:getValues() – returns a table holding the value/index of the rows that are currently selected.



## Visual Customization of the Picker Wheel

If you want to customize the appearance of the picker wheel, you can use the following commands. Customization is optional. You can use the default appearance as shown in Project 11.1.

- sheet – The image sheet object for the picker wheel.
- backgroundFrame – The frame number for the background that will sit behind the picker wheel. The image will be stretched to the full width and height of the picker wheel if needed.
- backgroundWidth, backgroundHeight – The width and height of the picker wheel background frame image.
- overlayFrame – frame number for the glass image or overlay.
- overlayFrameWidth, overlayFrameHeight – width and height of the picker wheel overlay frame image.
- separatorFrame – The frame number for the divider that separates each column.
- separatorWidth, separatorHeight – The width and height of the separator frame image.
- maskFile – A mask file used to crop picker wheel columns.

## Project 11.1 widget.newPickerWheel Example

In this example, a 3-column time picker wheel is created. First we will populate the variable minutes, which will be used in the second column to represent minutes, using a for loop. The for loop allows us to create the 0 to 59 elements of the array efficiently.

Next 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"

-- setup data that will be used in a column
local minutes = {}
for i=0,59 do
    minutes[i] = i
end

-- create table to hold all column data
local columnData = {
    { -- column 1
        labels = { "1", "2", "3", "4", "5", "6", "7", "8", "9", "10",
"11", "12" },
        align = "right",
        startIndex = 7,
    },
    { -- column 2
        labels = minutes,
        align = "center",
        startIndex = 30,
    },
    { -- column 3
        labels = { "AM", "PM" },
        startIndex = 2,
    }
}
```

Next we will create a picker wheel using 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
}
```

Finally, we will create a function to report the selected values when the button is tapped.

```
local function showValues( event )
    -- Retrieve the current values from the picker
```

```

local values = picker:getValues()

-- print the selected values
print( "Column 1: " .. values[1].value .. "Column 2: "
.. values[2].value .. "Column 3: " .. values[3].value)
end

local getValuesButton = widget.newButton
{
    left = 10,
    top = 150,
    width = 298,
    height = 56,
    id = "getValues",
    label = "Values",
    onRelease = showValues,
}

```

## **widget.newProgressView**

The progress view widget can be used in several ways, but is usually associated with either downloading files or showing the load progress of a file. In the example below, I have tied it to the slider widget so that as the slider's value is increased, the progress bar is updated to a new value. Note that the progress view is designed to increase. If you want to use it as a meter it will need to be reset each time the value changes. The progress view receives values between 0 and 1.

### **Parameters:**

- id – optional for specific progress view identification (default is “widget\_progressView”).
- left, top – specifies the top left corner for the progress view to be created.
- width – specifies the width of the progress view widget.
- isAnimated – optional, to animate the progress. Set to false for immediate updates to the value (default is false).
- fillXOffset – optionally allows you to position the horizontal offset of the fill image.
- fillYOffset – optionally allows you to position the vertical offset of the fill image.

### **Methods:**

- setProgress() – updates the progress view percentage. Receives a value between 0 and 1.

- getProgress() – returns the current value of the progress view.

### **Visual Customization of a Progress View:**

- sheet – The image sheet object for the progress view.
- fillOuterLeftFrame – The frame number for the outer left frame (the background/container).
- fillOuterMiddleFrame – The frame number for the outer middle frame (the background/container).
- fillOuterRightFrame – The frame number for the outer right frame (the background/container).
- fillInnerLeftFrame – The frame number for the inner left frame (the left edge of the fill).
- fillInnerMiddleFrame – The frame number for the inner middle frame (the middle of the fill).
- fillInnerRightFrame – The frame number for the inner right frame (the right edge of the fill).
- fillOuterWidth, fillOuterHeight – The width and height of the outer portion of the progress view frame.
- fillWidth, fillHeight – The width and height of the fill frames.

## **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 "widget\_slider").
- left, top – specifies the top left corner for the slider to be created.
- width, height – specifies the width and height of the slider widget.
- value – sets or returns value of the slider between 0 and 100 (default is 50).
- orientation – sets the orientation of the slider to vertical or horizontal (default is horizontal).
- listener - function called every time the slider is touched or moved.

### **Methods:**

- setValue() - returns the handle location by percentage (0 to 100).

## **Visual Customization of a Progress View:**

The slider can be customized in many ways depending on if its being used on a horizontal or vertical slider. The following properties are common to both horizontal and vertical sliders:

- sheet – The image sheet object for the progress view.
- frameWidth, frameHeight – the width and height of the slider edge frames.
- handleFrame – the handle/slider frame number.
- handleFrameWidth, handleFrameHeight – width and height of the handle frame.

## **Horizontal Slider Customization**

- leftFrame – left edge frame number for slider.
- middleFrame – middle frame number for slider.
- rightFrame – right edge frame number for slider.
- fillFrame – fill frame number for slider.

## **Vertical Slider Customization**

- topFrame – top edge frame number for slider.
- middleVerticalFrame – middle frame number for slider.
- bottomFrame – bottom frame number for slider.
- fillVerticalFrame – fill frame number for slider.

## **Project 11.2 Widget Slider & Progress View 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 graphics.

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 and updates the progress view. Remember that the progress view will only increase unless it is reset each time.



```
local widget = require("widget")

-----
-- widget.newProgressView()
-----

local newProgressView = widget.newProgressView
{
    left = 100,
    top = 150,
    width = 150,
    isAnimated = true,
}

    -- The listener for our slider
local function sliderListener( event )
    -- Update the progress view
    print(event.value)
    -- Must turn event.value into a number between 0 & 1.
    newProgressView:setProgress( event.value/100 )
end

-----
-- Create a horizontal slider
```

```
-----  
local horizontalSlider = widget.newSlider  
{  
    left = 100,  
    top = 232,  
    width = 150,  
    id = "Horizontal Slider",  
    listener = sliderListener,  
}
```

## widget.scrollView

The scroll view widget allows you to create scrolling content areas. It should be noted that if you want a scroll view 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 using graphics.newMask(). The scrollview is a fairly straight forward widget with only a few parameters:

### Parameters

- id – optional id assigned to the scroll view (default is “widget\_scrollView”)
- 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).
- scrollWidth, scrollHeight – required parameter giving the total scrollable area. Cannot be changed after the widget is created.
- topPadding, bottomPadding – optional number of pixels from the top and bottom when the scrolling reaches the end of the scrollable area (default is 0).
- leftPadding, rightPadding - optional number of pixels from the left and right when the scrolling reaches the end of the scrollable area.
- friction – determines how fast the rows travel when flicked up or down (default is 0.972).
- backgroundColor – optiona RGBA table for the background of the scroll view (default is white: {1, 1, 1, 1}).
- hideBackground – optional Boolean, hides the background of the scroll view area if set to true (default is false).
- horizontalScrollDisabled – optional Boolean parameter to disable horizontal scrolling.
- verticalScrollDisabled – optional Boolean parameter to disable vertical scrolling.
- isLocked – optional Boolean to keep the scroll view from scrolling.
- hideScrollBar – optional Boolean to hide the scroll bar from view.

- listener – function to listen for scroll view events. Provides two unique events:  
event.limitReached – when the scroll view reaches one of its limits; and  
event.direction – returns the direction that scroll view is moving.
- baseDir – optional way of setting the path to the mask file.
- maskFile – optional, used if a custom width and height are set for the scroll view.

## ScrollView Methods:

- scrollView:getContentPosition() – returns the current x, y position of the scroll view content. Used to mark current location (eg. local x, y = scrollView:getContentPosition() ).
- scrollView:scrollToPosition(table) – scroll to specified y position. Table options include x, y, time, and onComplete.
- scrollView:scrollTo(position, options) – scroll to top, bottom, left or right, with the options of time and onComplete.
- insert() – add items to scroll view.

## Visual Customization for Scroll View

Scroll view uses a scrollBarOptions table to pass visual customization information.

- sheet – image sheet containing the scroll bar images.
- topFrame – frame number for the top of the scrollBar.
- middleFrame – frame number for the middle of the scrollBar.
- bottomFrame – frame number for the bottom of the scrollBar.

We will look at an implementation of the scroll view in the tab bar example.

## **widget.newSegmentedControl**

The segmented control widget allows you to create a multi-part button that allows the user to select one of the options. Consider it the multiple choice test question of app design!



- id – optional id assigned to the segmented control (default is “widget\_segmentedControl”).
- left, top – allows custom location of segmented control (default is 0 for both values)
- width, height – sets the width and height of the segmented control’s frames. Each segment/frame must be the same width and height and will be loaded from the image sheet.
- segmentWidth – optional sets the segment width (default is 50 pixels).

- segments – a table containing the labels for each segment.
- defaultSegment – allows for the optional setting of a default/selected segment (if not specified, will default to the first segment).
- labelSize – font size for segment labels (default is 12).
- labelFont – font type for segment labels (default is native.systemFont).
- labelXOffset – optional horizontal offset for segment labels.
- labelYOffset – optional vertical offset for segment labels.
- onPress – optional function call when a segment is pressed.

### **Visual Customization of Segmented Control**

- sheet – the image sheet containing the images for your segmented control.
- leftSegmentedFrame, leftSegmentedSelectedFrame – default and pressed frame numbers for the left slice of the segmented control.
- middleSegmentedFrame, middleSegmentedSelectedFrame – default and pressed frame numbers for the middle slice of the segmented control.
- rightSegmentedFrame, rightSegmentedSelectedFrame - right default and pressed frame numbers for the right slice of the segmented control.
- dividerFrame – frame number for the divider line between each segment.

A segmented control demonstration is included in the tab bar example project.

### **widget.newSpinner**

The spinner widget is typically used to show that the application is busy; basically asking the user to be patient while the app performs needed functions.



#### **Parameters:**

- id – optional id assigned to the spinner (default is “widget\_spinner”).
- left\_top – allows custom location of spinner (default is 0 for both values).
- width, height – optional, sets the width and height of the spinner frames if you are using a custom imageSheet.
- time – optional, set the time for the spinners animation (default is 1000 milliseconds).
- deltaAngle – optional, sets the delta angle for the spinner rotation per increment.
- incrementEvery – optional, the delay in milliseconds between each segment rotation.

#### **Methods:**

- start()- begins the animation of the spinner widget.

- stop() – you guessed it! Stops the spinner animation.

### Visual Customization of Spinner

- sheet – the image sheet containing the images for your spinner.
- startFrame – the frame number for the spinners first frame.
- frameCount – the number of frames in the spinner animation (default is 1).

A spinner demonstration is included in the tab bar example project.

### **widget.newStepper**

The stepper is a simple widget for incrementing or decrementing a value.



#### Parameters:

- id – optional id assigned to the stepper (default is “widget\_stepper”).
- left, top – allows custom location of stepper (default is 0 for both values).
- width, height – sets the width and height of the stepper frames if you are using a custom imageSheet.
- initialValue – sets the initial value for stepper (default is 0).
- minimumValue – optionally sets the minimum value that stepper can decrement to (default is 0).
- maximumValue – optionally sets the maximum value that stepper can increment to.
- onPress – optional function to be called with stepper is pressed. Includes possible event.phases of increment, decrement, minLimit, maxLimit.
- onHold – optional function call while stepper segment is being held down. Includes possible event.phases of increment, decrement, minLimit, maxLimit.
- value – returns the current value.

### Visual Customization of Stepper

- sheet – the image sheet containing the images for your stepper.
- defaultFrame – frame number for steppers default (both + and - active).
- noMinusFrame – frame number for minLimit frame (+ active, - greyed out).
- noPlusFrame – frame number for maxLimit frame (+ greyed out, - active).
- minusActiveFrame – frame number with minus pressed or held (- in pressed state).
- plusActiveFrame – frame number with plus pressed or held (+ in pressed state).

A stepper demonstration is included in the tab bar example project.

## **widget.newSwitch**

The switch widget is a very flexible tool, able to be displayed as an on/off switch, checkbox or radio button.



### **Parameters:**

- id – optional id assigned to the switch (default is “widget\_switch”).
- left, top – allows custom location of switch (default is 0 for both values).
- width, height – sets the width and height of the switch frames if you are using a custom imageSheet.
- initialSwitchState – sets the initial value for switch (default is false/off/deselected).
- style – optionally sets the style of the switch. Options are radio, checkbox and onOff. (default is onOff).
- onPress – optional function to be called when the switch is pressed.
- onRelease – optional function call when the user releases the switch.
- onEvent – optional function call that should only be used if onPress or onRelease are not used.

### **Methods:**

- setState(options) – used to set the state of the switch. Options includes the table items *isOn*, *isAnimated*, *onComplete*.

## **Visual Customization of Switch**

Switch has several possible visual customization options including Radio, Checkbox and on/off switch. The sheet parameter is used by each of these options.

- sheet – the image sheet containing the images for your switch.

## **Visual Customization of Radio & Checkbox Switch**

- frameOff – frame number for switch off.
- frameOn – frame number for switch on.

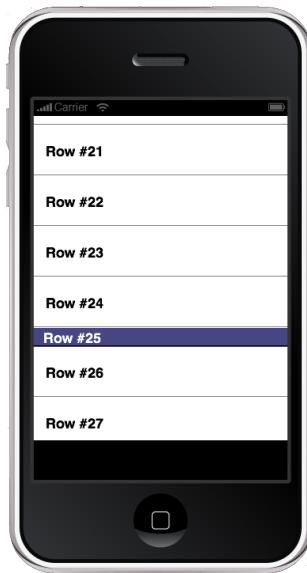
## **Visual Customization of On/Off Switch**

- onOffBackgroundFrame – frame number for background frame (frame is the image that shows the two colors and on/off text).
- onOffBackgroundWidth, onOffBackgroundHeight – frame width and height for image in custom imageSheet.
- onOffOverlayWidth, onOffOverlayHeight – frame width and height for overlay.
- onOffOverlayFrame – frame number for switch overlay.
- onOffHandleDefaultFrame – the frame number for the handle.
- onOffHandleOverFrame – the frame number for the handle over frame.

The switch will be demonstrated in the tab bar project example.

### **widget.newTableView**

The tableView widget allows you to create scrolling lists. This is an example of the third type of table discussed in Chapter 10. With this widget you can control the rendering of the individual rows. You might notice that you are actually rendering or building each row for view. This gives you a great deal of control of what to include on each row including images and/or text. The tableview widget is very powerful and flexible; with that flexibility comes a few more properties and methods:



Parameters:

- [id](#) – optional id assigned to the table view (default is “widget\_tableView”).
- [backgroundColor](#) – RGBA table to set the color of the rectangle that is behind the tableView (default is white {1, 1, 1, 1}).
- [left, top](#) – allows custom location of table view (default is 0).
- [width, height](#) – allows custom width and height of table view (default width is full width and height of the screen).
- [friction](#) – determines how fast the rows travel when flicked up or down (default is 0.972).
- [maskFile](#) – allows a custom height through the use of a masking file.
- [hideBackground](#) – optional Boolean to hide the background of the widget.

- 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).
- maxVelocity – optional limit to the maximum scrolling speed (default is 2).
- noLines – optional method of hiding lines separating rows.
- isLocked – optional method of locking the table view so that it cannot scroll vertically.
- hideScrollBar – optional method of hiding the scrollbar.
- onRowRender – optional function call on the initial rendering of the table.
- onRowUpdate – optional function call for when previously viewed rows become visible again.
- onRowTouch – optional function call for when a row is touched.
- listener – optional function call to handle all table events.
- baseDir – optional to set directory for loading the maskFile.

#### **Methods:**

- getContentPane() – returns the current y position of the table view content. Used to mark current location.
- scrollToY(y position, time) – scroll to specified y position. Time is in milliseconds for how long it takes to scroll to location (default time is 1500).
- 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).
- insertRow ({parameters}) – used to insert rows into the table view. Accepts as parameters:

#### **insertRow Parameters:**

- 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 specifying the current row as a category.
- deleteRow ( row or index) – deletes a specific row.
- deleteAllRows() – you probably guess, it deletes all the rows of the table view.

**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 by setting up our listeners. The first one will print to the terminal window all table events that occur.

```
local widget = require( "widget" )

-- Listen for tableView events
local function tableViewListener( event )
    local phase = event.phase

    print( event.phase )
end
```

The next two listener functions handle reporting when a row reappears on the screen and reporting when a row is touched. Don’t worry; I have a more sophisticated example of using table view later in the book that will show how to create tables that go several screens deep.

```
-- Handle row's becoming visible on screen
local function onRowUpdate( event )
    local row = event.row

    print( "Row:", row.index, " is now visible" )
end

-- Handle touches on the row
local function onRowTouch( event )
    local phase = event.phase
```

```

if "press" == phase then
    print( "Touched row:", event.target.index )
end

```

The `onRowRender` function handles creating the row for the display as the user scrolls through the list.

```

-- Handle row rendering
local function onRowRender( event )
    local phase = event.phase
    local row = event.row

    local rowTitle = display.newText( row, "Row " .. row.index,
0, 0, nil, 14 )
    rowTitle.x = row.x - ( row.contentWidth * 0.5 ) + (
rowTitle.contentWidth * 0.5 )
    rowTitle.y = row.contentHeight * 0.5
    rowTitle:setFillColor( 0, 0, 0 )
end

```

Time to create our table view widget. I have included the `mask-410.png` file in the resource folder for Chapter 11.

```

-- Create a tableView
local tableView = widget.newTableView
{
    top = 100,
    width = 320,
    height = 410,
    maskFile = "mask-410.png",
    listener = tableViewListener,
    onRowRender = onRowRender,
    onRowTouch = onRowTouch,
}

```

Now we will create the row information. In the 25<sup>th</sup> and 45<sup>th</sup> rows, categories are created. The final portion of the code handles inserting the row and setting the events and render handlers.

```
-- Create 100 rows
```

```

for i = 1, 100 do
    local isCategory = false
    local rowHeight = 40
    local rowColor =
    {
        default = { 1, 1, 1 },
    }
    local lineColor = { .8, .8, .8 }

    -- Make some rows categories
    if i == 25 or i == 50 or i == 75 then
        isCategory = true
        rowHeight = 24
        rowColor =
        {
            default = { .4, .5, .55, .9 },
        }
    end
    -- Insert the row into the tableView
    tableView:insertRow
    {
        isCategory = isCategory,
        rowHeight = rowHeight,
        rowColor = rowColor,
        lineColor = lineColor,
    }
end

-- delete the tenth row in the tableView
tableView:deleteRow( 10 )

```

## **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:**

- `id` – string to identify button (default is “`widget tabBar`”).
- `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)
- `buttons` – table holding the parameters and options for each tab button (see Buttons Table).



### Method:

- `setSelected(buttonIndex, simulatePress)` – makes a specific button selected. Provide the index number of the button to show as pressed. If you pass a true in the `simulatePress` parameter, the app will call the function supplied on `onPress`.

### Buttons Table

- `id` – optional string to identify button (default is “button”).
- `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).
- `onPress` – function called when the button is tapped.
- `selected` – Boolean to track if the button is selected (down). Only one button may be down at a time.
- `labelXOffset, labelYOffset` – optional horizontal and vertical offset to tab bar label.

Buttons can be created from either image files or from an image sheet.

## **Button Creation from Image Files**

- width, height - should match the width/height of your defaultFile/overFile.
- baseDir – optional, sets the base directory where your custom images are located (default is your project folder system.ResourceDirectory).
- defaultFile - the default ("un-pressed") image of the TabBar button.
- overFile -the over ("pressed") image of the TabBar button.

## **Button Creation from Image Sheet**

- defaultFrame - the default ("un-pressed") frame index of the TabBar button.
- overFrame - the over ("pressed") frame index of the TabBar button.

## **Customized ImageSheet Tab Bar Creation**

- sheet - the image sheet object for your tab bar.
- backgroundFrame - the background frame number of the tab bar.
- backgroundFrameWidth - the width/height of the background image of the tab bar.
- tabSelectedLeftFrame - the left edge frame number of the tab bar's selected graphic.
- tabSelectedMiddleFrame - the left edge frame number of the tab bar's selected graphic.
- tabSelectedRightFrame -the left edge frame number of the tab bar's selected graphic.
- tabSelectedFrameWidth, tabSelectedFrameHeight -the width/height of the tab bar selected graphic.

## **Project 11.4 Widget Tab Bar Example**

Since the tab bar was designed to show multiple views or screens, I thought it would be a perfect method to show some of the less complex widgets. In this project, I will use composer to demonstrate the tab bar, which will be used for the calls to show the scroll bar, spinner, stepper, switch, and segmented control widgets. This project is similar to Corona Labs Widget Demo (giving credit where it is due!).

### **main.lua**

To begin with we will setup a background and load the required widget and composer APIs. We will also setup a gradient for the title bar so that it looks a little flashier.

```
-- Set the background to white
```

```

display.setDefault( "background", 1, 1, 1 )

-- Require the widget & composer libraries
local widget = require( "widget" )
local composer = require( "composer" )

-- The gradient used by the title bar
local titleGradient = graphics.newGradient(
    { .55, .8, .9, 1 },
    { .2, .3, .4, 1 }, "down" )

-- Create a title bar
local titleBar = display.newRect( 0, 0, display.contentWidth, 32 )
titleBar.y = titleBar.contentHeight * 0.5
titleBar:setFillColor( titleGradient )

-- Create the title bar text
local titleBarText = display.newText( "Widget Demo", 0, 0,
native.systemFontBold, 16 )
titleBarText.x = titleBar.x
titleBarText.y = titleBar.y

```

Now we are ready to setup the icon bar. I kept it very simple loading image files for each of the 3 tabs.

```

-- Create buttons table for the tab bar
local tabButtons =
{
    {
        width = 32,
        height = 32,
        defaultFile = "tabIcon.png",
        overFile = "tabIcon-down.png",
        label = "Segemented",
        onPress = function()composer.gotoScene( "tab1" ); end,
        selected = true
    },
    {
        width = 32,

```

```

        height = 32,
        defaultFile = "tabIcon.png",
        overFile = "tabIcon-down.png",
        label = "ScrollView",
        onPress = function()composer.gotoScene( "tab2" ) ; end,
    },
{
    width = 32,
    height = 32,
    defaultFile = "tabIcon.png",
    overFile = "tabIcon-down.png",
    label = "Other",
    onPress = function()composer.gotoScene( "tab3" ) ; end,
}
}

```

After defining the tab bar, we can use the definition to create the tabBar widget. Our last step of the main.lua will be to transfer control to tab1.lua.

```

-- Create a tab-bar and place it at the bottom of the screen
local tabBar = widget.newTabBar
{
    top = display.contentHeight - 50,
    width = display.contentWidth,
    buttons = tabButtons
}

-- Start at tab1
composer.gotoScene( "tab1" )

```

## **tab1.lua**

In tab1.lua, we are going to use the segmented control to output the select segment to a status display box. To get things started, we will load widget and composer, and then create a scene to store this composer scene.

```

local widget = require( "widget" )
local composer = require( " composer " )
local scene = composer.newScene()

-- Our scene

```

```

function scene:create(event )
    local group = self.view

        -- Display a background
    local background = display.newImage( "background.png", true
)
    group:insert( background )

```

Here we will make a simple rectangle into a status update box. This is a simple method to display changes in the segmented control to the screen so that the user can see what is happening.

```

-- Status text box
local statusBox = display.newRect( 70, 290, 210, 120 )
statusBox:setFillColor( 0, 0, 0 )
statusBox.alpha = 0.4
group:insert( statusBox )

-- Status text
local statusText = display.newText( "Interact with a widget to begin!", 80, 300, 200, 0, native.systemFont, 20 )
statusText.x = statusBox.x
statusText.y = statusBox.y - ( statusBox.contentHeight * 0.5 ) + ( statusText.contentHeight * 0.5 )
group:insert( statusText )

-----
-- widget.newSegmentedControl()
-----
```

Next we will setup a function that when one of the segmented controls is tapped, it will pass which one to the status box.

```

-- The listener for our segmented control
local function segmentedControlListener( event )
    local target = event.target

        -- Update the status box text
    statusText.text = "Segmented Control\nSegment Pressed: "
.. target.segmentLabel

        -- Update the status box text position
    statusText.x = statusBox.x
    statusText.y = statusBox.y - ( statusBox.contentHeight * 0.5 ) + ( statusText.contentHeight * 0.5 )
```

```
end -- end of segmentedControlListener function
```

Time to create the segmented control:

```
-- Create a default segmented control
local segmentedControl = widget.newSegmentedControl
{
    left = 10,
    top = 60,
    segments = { "Aren't", "Segment", "Control", "Widgets",
"Fun?" },
    defaultSegment = 1,
    onPress = segmentedControlListener,
}
group:insert( segmentedControl )
end
```

And that ends our create scene function. Now we just need to call it and return control back to the composer api.

```
scene:addEventListener( "create" )

return scene
```

## tab2.lua

In tab2 we will use the scrollView widget to handle a large picture. As before, we will start by loading the required APIs and setup the scene.

```
local widget = require( "widget" )
local composer = require( "composer" )
local scene = composer.newScene()

-- Our scene
function scene:create( event )
    local group = self.view

    -- Display a background
    local background = display.newImage( "background.png", true
)
    group:insert( background )
```

Now for the ScrollView Listener function, which will just report to the terminal any event phases that occur.

```

-- Our ScrollView listener
local function scrollListener( event )
    local phase = event.phase
    local direction = event.direction

    if "began" == phase then
        --print( "Began" )
    elseif "moved" == phase then
        --print( "Moved" )
    elseif "ended" == phase then
        --print( "Ended" )
    end

    -- If the scrollView has reached it's scroll limit
    if event.limitReached then
        if "up" == direction then
            print( "Reached Top Limit" )
        elseif "down" == direction then
            print( "Reached Bottom Limit" )
        elseif "left" == direction then
            print( "Reached Left Limit" )
        elseif "right" == direction then
            print( "Reached Right Limit" )
        end
    end

    return true
end

```

Now to create the scroll view, loading the mask for the image and setting up the listener.

```

-- Create a ScrollView
local scrollView = widget.newScrollView
{
    left = 10,
    top = 52,
    width = 300,
    height = 350,
    id = "onBottom",
    hideBackground = true,
    horizontalScrollingDisabled = false,
    verticalScrollingDisabled = false,
    maskFile = "scrollViewMask-350.png",
    listener = scrollListener,
}

```

Finally, load the image and position it on the screen.

```
-- Insert an image into the scrollView
local background = display.newImageRect( "scrollimage.jpg",
768, 1024 )
background.x = background.contentWidth * 0.5
background.y = background.contentHeight * 0.5
scrollView:insert( background )
group:insert( scrollView )
end

scene:addEventListener( "create" )

return scene
```

Surprisingly simple! Let's finish up with tab3, which includes the spinner, stepper, and 3 types of switches.

### **tab3.lua**

As before, we will setup the composer structure and use the status box as like we did in tab1.lua.

```
local widget = require( "widget" )
local composer = require( "composer" )
local scene = composer.newScene()

-- Our scene
function scene:create( event )
    local group = self.view

        -- Display a background
    local background = display.newImage( "background.png", true )
    group:insert( background )

        -- Status text box
    local statusBox = display.newRect( 70, 290, 210, 120 )
    statusBox:setFillColor( 0, 0, 0 )
    statusBox.alpha = 0.4
    group:insert( statusBox )

        -- Status text
```

```

local statusText = display.newText( "Interact with a widget
to begin!", 80, 300, 200, 0, native.systemFont, 20 )
statusText.x = statusBox.x
statusText.y = statusBox.y - ( statusBox.contentHeight *
0.5 ) + ( statusText.contentHeight * 0.5 )
group:insert( statusText )

```

Since we are not using a custom spinner widget, adding the spinner is as easy as creating it and telling it where you want it located.

```

-----
-- widget.newSpinner()
-----

-- Create a spinner widget
local spinner = widget.newSpinner
{
    left = display.contentWidth/2,
    top = 55,
}
group:insert( spinner )

-- Start the spinner animating
spinner:start()

```

Time to add the stepper widget. We will setup a simple text and increment/decrement according to the users touches.

```

-----
-- widget.newStepper()
-----

-- Create some text for the stepper
local currentValue = display.newText( "Value: 00", 165,
105, native.systemFont, 20 )
currentValue:setFillColor( 0 )
group:insert( currentValue )

-- The listener for our stepper
local function stepperListener( event )
    local phase = event.phase

```

```

    -- Update the text to reflect the stepper's current
value
    currentValue.text = "Value: " .. string.format( "%02d",
event.value )
end

```

Now to add the stepper widget. As you can see, it is a simple widget to use!

```

-- Create a stepper
local newStepper = widget.newStepper
{
    left = 50,
    top = 105,
    initialValue = 0,
    minimumValue = 0,
    maximumValue = 50,
    onPress = stepperListener,
}
group:insert( newStepper )

```

Time for the switch widget. We are going to do all three types, so there will be a little bit of programming to do here. First, we will setup the listener to update the status box text depending on what the user taps.

```

-----
-- widget.newSwitch()
-----

-- The listener for our radio switch
local function radioSwitchListener( event )
    -- Update the status box text
    statusText.text = "Radio Switch\nIs on?: " ..
tostring( event.target.isOn )

    -- Update the status box text position
    statusText.x = statusBox.x
    statusText.y = statusBox.y - ( statusBox.contentHeight
* 0.5 ) + ( statusText.contentHeight * 0.5 )
end

-- Create some text to label the radio button with

```

```

local radioButtonText = display.newText( "Use?", 40, 150,
native.systemFont, 16 )
radioButtonText:setFillColor( 0 )
group:insert( radioButtonText )

```

The Radio Button. Once the style is declared, it is just a matter of setting the switch state and the listener, and it is ready to go!

```

-- Create a default radio button (using widget.setTheme)
local radioButton = widget.newSwitch
{
    left = 25,
    top = 180,
    style = "radio",
    id = "Radio Button",
    initialSwitchState = true,
    onPress = radioSwitchListener,
}
group:insert( radioButton )

local otherRadioButton = widget.newSwitch
{
    left = 55,
    top = 180,
    style = "radio",
    id = "Radio Button2",
    initialSwitchState = false,
    onPress = radioSwitchListener,
}
group:insert( otherRadioButton )

-- Create some text to label the checkbox with
local checkboxText = display.newText( "Sound?", 110, 150,
native.systemFont, 16 )
checkboxText:setFillColor( 0 )
group:insert( checkboxText )

-- The listener for our checkbox switch
local function checkboxSwitchListener( event )
    -- Update the status box text
    statusText.text = "Checkbox Switch\nIs on?: " ..
tostring( event.target.isOn )

    -- Update the status box text position
    statusText.x = statusBox.x

```

```

        statusText.y = statusBox.y - ( statusBox.contentHeight
* 0.5 ) + ( statusText.contentHeight * 0.5 )
    end

```

As you can see, the checkbox is even easier than the radio button (since the radio button requires 2 or more button objects).

```

-- Create a default checkbox button (using widget.setTheme)
local checkboxButton = widget.newSwitch
{
    left = 120,
    top = 180,
    style = "checkbox",
    id = "Checkbox button",
    onPress = checkboxSwitchListener,
}
group:insert( checkboxButton )

-- Create some text to label the on/off switch with
local switchText = display.newText( "Music?", 200, 150,
native.systemFont, 16 )
switchText:setFillColor( 0 )
group:insert( switchText )

```

And finally the on/off switch. Like the other switches, it only requires a few parameters to take full use and make your life a lot easier.

```

-- The listener for our on/off switch
local function onOffSwitchListener( event )
    -- Update the status box text
    statusText.text = "On/Off Switch\nIs on?: " ..
tostring( event.target.isOn )

    -- Update the status box text position
    statusText.x = statusBox.x
    statusText.y = statusBox.y - ( statusBox.contentHeight
* 0.5 ) + ( statusText.contentHeight * 0.5 )
end

-- Create a default on/off switch (using widget.setTheme)
local onOffSwitch = widget.newSwitch
{
    left = 190,
    top = 180,
    initialSwitchState = true,

```

```

        onPress = onOffSwitchListener,
        onRelease = onOffSwitchListener,
    }
    group:insert( onOffSwitch )
end

scene:addEventListener( "create" )

return scene

```

## 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.

## Summary

Once you become more familiar with using widgets, you will find that they greatly increase your productivity and speed in creating applications. Obviously it will take some practice to become familiar with some of the features available through these apps. To help you gain experience, we will be using many of these widgets throughout the remainder of the textbook.

## Assignments

1. Create an app that displays a number between 1 and 100 that updates as the user moves a slider.
2. Rewrite the age app from chapter 4 to use a picker wheel instead of textfields.
3. 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).

4. Select one of the previous apps that you have made and replace all buttons in the app with the button widget.
5. (Challenge) Create your own personalized widget theme.

# Chapter 12: System Events & Tools

## Learning Objectives

When developing apps to sell on the app stores, it is good programming practice to develop your app to properly handle things going wrong (of course, it won't be YOUR app that messes up ;-). In this chapter we will examine some of the system events and system resources that are available. We will examine:

- How to handle system events
- How to use the accelerometer
- How to use the gyroscope
- How to use the GPS/Compass
- How to use maps in Corona
- How to use Alerts for notification

## System Events

If you hadn't noticed, your smartphone and tablet are pretty amazing devices. Most of them have many built in features that make creating sophisticated apps pretty easy.

System event are events that are sent by the smartphone or tablet and broadcast for any interested listener to respond. Some key system events that we have previously discussed include orientation change and enterFrame.

A very important system event that you should prepare for is the closing of your app. Whether the user is receiving a phone call, instant message, or just closing the app to use something else, most apps need to save their data so that they can properly resume. The system will notify your app if any of these cases occur so that it can properly save and can resume in the future. A sample way to handle such an event is:

```
local function onSystemEvent(event)
    -- handle unexpected close or interruptions
    if (event.type=="applicationExit" or event.type=="applicationSuspend") then
        -- save app information (see chapter 13 on how to save)
        saveStatus()
    elseif(event.type=="applicationResume") then
        loadStatus()
    end
Runtime:addEventListener( "system", onSystemEvent )
```

## Accelerometer

Most modern smart phones and many tablet devices are able to supply accelerometer data (movement in a specific direction) to the app developer. Corona has simple event API resources for each.

The accelerometer is able to measure movement in 3 dimensions. This includes registering shake events and movement in the x, y, or z directions. To use the accelerometer you will need to setup a runtime event:

```
Runtime:addEventListener("accelerometer", listenerFunction)
```

Properties available through the accelerometer API include:

- event.isShake – returns true if the device is being shook.
- event.deltaTime – returns the amount of time in seconds since the last accelerometer event.
- event.xGravity - returns the amount of acceleration in the x direction due to gravity.
- event.yGravity – returns the amount of acceleration in the y direction due to gravity.
- event.zGravity – returns the amount of acceleration in the z direction due to gravity.
- event.xInstant – returns the instant acceleration in the x direction.
- event.yInstant – returns the instant acceleration in the y direction.
- event.zInstant – returns the instant acceleration in the z direction.
- system.setAccelerometerInterval() – set frequency of accelerometer events in Hertz (cycles per second). Min. is 10, max 100. Lower is better to conserve battery life.

Wow, that is about as clear as mud. Let's take a look at its use in an app to try to clarify the accelerometer properties. One warning before we get started, the accelerometer does not work in the Corona simulator (beyond shake). Moving the simulator window around your screen will not produce an accelerometer event (it was a nice try though). The only accelerometer event that you can simulate in the simulator is the shake event. To see the accelerometer in action you will have to build the app and deploy it to a device.

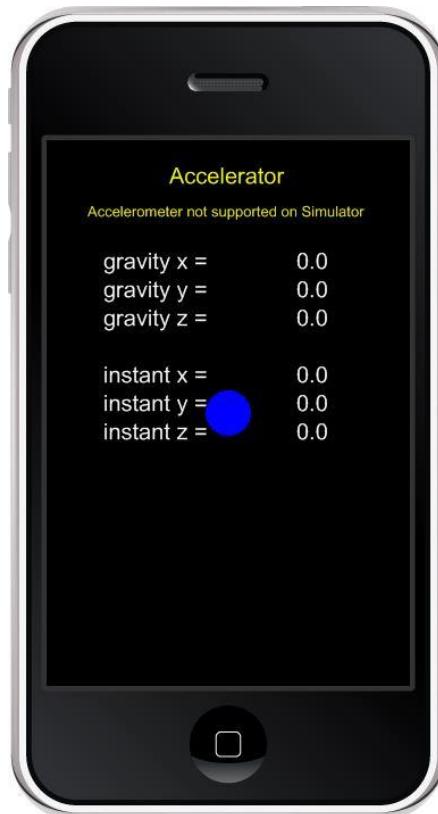
Let's talk about gravity vs. instant in the API information above. Gravity is the continuous pull of gravity on your device. If you hold your device in the customary portrait view, the primary gravity variable changed will be yGravity. If you tip your device into landscape

view, it will be xGravity. And if you lay your phone down (so that you can see the screen, it's kind of pointless to do it the other way), zGravity is what is affected.

Instant is the detection of movement in a direction. If you move the device side to side (assuming you are back in portrait), you will see the xInstant change. If you move it up and down, then it will be yInstant that changes. And if you pretend you are in Sesame Street and do the "Near and Far" routine, zInstant will change.

## Project 12.0 Accelerometer

So let's look at an implementation of an app that will show how to use these tools. I developed this app based upon the Corona sample app. It has been greatly simplified, which will hopefully help you to see the important bits.



We will get started by hiding the status bar and putting a title at the top of the screen. Next we will test to see if the app is being run on the simulator or on a device.

```
display.setStatusBar(display.HiddenStatusBar) -- hide status bar  
-- Displays App title  
title = display.newText( "Accelerator", 0, 20, nil, 20 )  
title.x = display.contentWidth/2 -- center title
```

```

title:setFillColor( 1, 1, 0 )

-- Determine if running on Corona Simulator
local isSimulator = "simulator" == system.getInfo("environment")

-- Accelerator is not supported on Simulator
if isSimulator then
    msg = display.newText( "Accelerometer not supported on
Simulator", 0, 55, nil, 13 )
    msg.x = display.contentWidth/2           -- center title
    msg:setFillColor( 1,1,0 )
end

local soundID = audio.loadSound ("beep_wav.wav")

```

Next we will setup the text to show changes in gravity and instant accelerometer feedback. As you can see, I am incrementing y to make it easier to place the text on the screen.

```

local y = 95

local xgText = display.newText( "gravity x = ", 50, y, nil, 20 )
xgText:setFillColor(1, 1, 1)
local xg = display.newText( "0.0", 220, y, nil, 20 )
xg:setFillColor(1, 1, 1)
y = y + 25
local ygText = display.newText( "gravity y = ", 50, y, nil, 20 )
local yg = display.newText( "0.0", 220, y, nil, 20 )
ygText:setFillColor(1, 1, 1)
yg:setFillColor(1, 1, 1)
y = y + 25
local zgText = display.newText( "gravity z = ", 50, y, nil, 20 )
local zg = display.newText( "0.0", 220, y, nil, 20 )
zgText:setFillColor(1, 1, 1)
zg:setFillColor(1, 1, 1)
y = y + 50
local xiText = display.newText( "instant x = ", 50, y, nil, 20 )
local xi = display.newText( "0.0", 220, y, nil, 20 )
xiText:setFillColor(1, 1, 1)
xi:setFillColor(1, 1, 1)
y = y + 25
local yiText = display.newText( "instant y = ", 50, y, nil, 20 )
local yi = display.newText( "0.0", 220, y, nil, 20 )
yiText:setFillColor(1, 1, 1)
yi:setFillColor(1, 1, 1)
y = y + 25
local ziText = display.newText( "instant z = ", 50, y, nil, 20 )

```

```
local zi = display.newText( "0.0", 220, y, nil, 20 )
ziText:setFillColor(1, 1, 1)
zi:setFillColor(1, 1, 1)
```

To make the app a little more visually appealing, we will include a circle that will eventually be impacted by the gravity property.

```
-- Create a circle that moves with Accelerator events (for
visual effects)

local centerX = display.contentWidth / 2
local centerY = display.contentHeight / 2

Circle = display.newCircle(0, 0, 20)
Circle.x = centerX
Circle.y = centerY
Circle:setFillColor( 0, 0, 1 )           -- blue
```

The next function receives the information from the accelerometer event, receiving which object event happened (xGravity, yGravity, zGravity, xInstant, yInstant, or zInstant), and the new value for that object. While we have mentioned string.format previously, this is an example of using the string format to convert the value supplied by the accelerator and converting it into a value that is understandable. The %1.3f tells formats the text to have at least 1 digit before the decimal and up to 3 after the decimal places in precision. The f tells the app that it is a floating point variable (i.e. it has decimal places).

```
-----
-- Hardware Events
-----

-- Display the Accerator Values
local function xyzFormat( obj, value)

    obj.text = string.format( "%1.3f", value )

end

-- Called for Accelerator events
-- Update the display with new values
```

Our final function calls the formatting function above, passing the object variable to update and the value based upon the event property. The last step before creating the event

listener is to update the circle's location based upon changes in gravity (i.e. the tilt of the phone).

```
local function onAccelerate( event )

    -- Format and display the Accelerator values
    --
    xyzFormat( xg, event.xGravity)
    xyzFormat( yg, event.yGravity)
    xyzFormat( zg, event.zGravity)
    xyzFormat( xi, event.xInstant)
    xyzFormat( yi, event.yInstant)
    xyzFormat( zi, event.zInstant)

    -- Move our object based on the accelerator values
    Circle.x = centerX + (centerX * event.xGravity)
    Circle.y = centerY + (centerY * event.yGravity * -1)
end

-- Add runtime listener
Runtime:addEventListener ("accelerometer", onAccelerate);
```

If you deploy this app to your own device, you will notice that it is actually very sensitive. Minor movements and shifts in how you hold your device have a real impact on the numbers reported.

## Gyroscope

The availability of the gyroscope will depend on its availability on the devices that it is deployed to. To use the gyroscope function on iOS devices, you will need to turn it on in the build.settings file with the command:

```
settings =
{
    iphone =
    {
        plist =
        {
            UIRequiredDeviceCapabilities = "gyroscope"
        },
    },
}
```

```
}
```

The properties for the gyroscope include:

- event.xRotation – rotation around the devices x axis in radians per second.
- event.yRotation - rotation around the devices y axis in radians per second.
- event.zRotation - rotation around the devices z axis in radians per second.
- event.deltaTime – time in seconds since last gyroscope event.
- system.setGyroscopeInterval() – sets the frequency of gyroscope updates in Hertz (cycles per second). Minimum value is 10, max 100. Lower is better as it conserves battery life.

As with the accelerometer API, gyroscope will not work in the simulator, so you will have to build the next project and deploy it for it to work.

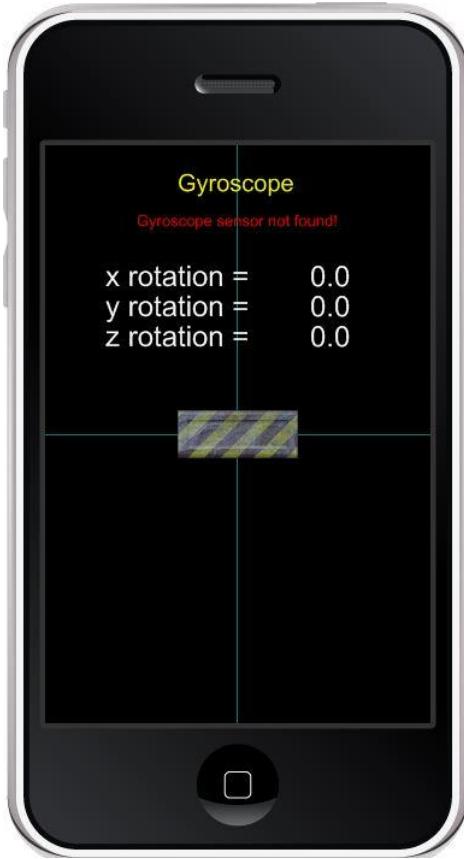
## Project 12.1 Gyroscope

Using similar code as in the accelerometer project, we will now look at how to use the gyroscope. As before, this is a greatly simplified version of the demonstration that comes in the Corona sample code folder.

To get started, we will activate gyroscope for iOS devices in the build.settings file.

build.settings

```
-- cpmgen build.settings
settings =
{
    iphone =
    {
        plist =
        {
            UIRequiredDeviceCapabilities = "gyroscope"
        },
    },
}
```



As in the accelerometer project, we will start by setting up our screen, which includes a title, x & y axes lines, and information for the user as they rotate their device.

### main.lua

```
-- Project: Ch 12.1 Gyroscope main.lua

-- Hide the status bar.
display.setStatusBar(display.HiddenStatusBar)

-- Draw X and Y axes.
local xAxis = display.newLine(0, display.contentHeight / 2,
display.contentWidth, display.contentHeight / 2)
xAxis:setStrokeColor( 0, 1, 1, .5 )
local yAxis = display.newLine(display.contentWidth / 2, 0,
display.contentWidth / 2, display.contentHeight)
yAxis:setStrokeColor( 0, 1, 1, .5 )

-- Displays App title
local title = display.newText( "Gyroscope", 0, 20, nil, 20 )
```

```

title.x = display.contentWidth / 2
title:setFillColor( 1, 1, 0 )

-- Notify the user if the device does not have a gyroscope.
if not system.hasEventSource("gyroscope") then
    local msg = display.newText( "Gyroscope sensor not found!",
0, 55, nil, 13 )
    msg.x = display.contentWidth / 2
    msg:setFillColor( 1, 0, 0 )
end
-----
-- Create Text and Display Objects
-----

-- Text parameters
local x = 220
local y = 95

local xHeaderLabel = display.newText( "x rotation = ", 50, y,
nil, 24 )
xHeaderLabel:setFillColor(1, 1, 1)
local xValueLabel = display.newText( "0.0", x, y, nil, 24 )
xValueLabel:setFillColor(1, 1, 1)
y = y + 25

local yHeaderLabel = display.newText( "y rotation = ", 50, y,
nil, fontSize )
local yValueLabel = display.newText( "0.0", x, y, nil, 24 )
yHeaderLabel:setFillColor(1, 1, 1)
yValueLabel:setFillColor(1, 1, 1)
y = y + 25

local zHeaderLabel = display.newText( "z rotation = ", 50, y,
nil, 24 )
local zValueLabel = display.newText( "0.0", x, y, nil, 24 )
zHeaderLabel:setFillColor(1, 1, 1)
zValueLabel:setFillColor(1, 1, 1)

```

As with most things, having a visual representation of what is happening is always nice. We will load a box that will demonstrate the rotations.

```
-- Create an object that moves with gyroscope events.
local centerX = display.contentWidth / 2
local centerY = display.contentHeight / 2
target = display.newImage("target.png", true)
target.x = centerX
target.y = centerY
```

Now we will handle the events for the gyroscope much like we did for the accelerometer.

```
-----
-- Hardware Events
-----

-- Display the Gyroscope Values
local function xyzFormat( obj, value )

    obj.text = string.format( "%1.3f", value )
end

-- Called when a gyroscope measurement has been received.
local function onGyroscopeUpdate( event )

    -- Format and display the measurement values.
    xyzFormat(xValueLabel, event.xRotation)
    xyzFormat(yValueLabel, event.yRotation)
    xyzFormat(zValueLabel, event.zRotation)

    -- Move our object based on the measurement values.
    local nextX = target.x + event.yRotation
    local nextY = target.y + event.xRotation
    if nextX < 0 then
        nextX = 0
    elseif nextX > display.contentWidth then
        nextX = display.contentWidth
    end
    if nextY < 0 then
        nextY = 0
    elseif nextY > display.contentHeight then
        nextY = display.contentHeight
    end
```

```

target.x = nextX
target.y = nextY

-- Rotate the graphic box based on the degrees rotated
around the z-axis.
local deltaRadians = event.zRotation * event.deltaTime
local deltaDegrees = deltaRadians * (180 / math.pi)
target:rotate(deltaDegrees)
end

-- Add gyroscope listeners, but only if the device has a
gyroscope.
if system.hasEventSource("gyroscope") then
    Runtime:addEventListener("gyroscope", onGyroscopeUpdate)
end

```

Obviously, the device must have gyroscope capabilities to make use of this app. One of my test devices (an old Nexus One) was not able to perform the operations since it doesn't have this built in capability.

## Alerts

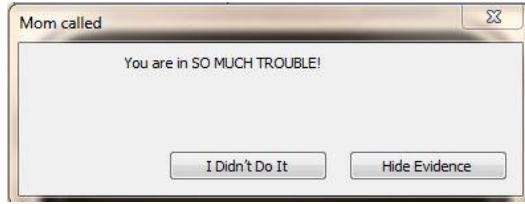
Sometimes when we are creating an app, we need to notify the user of a problem or provide information that they should not ignore. `native.showAlert()` displays a popup alert box with one or more buttons. The app will remain active in the background but all of the user's activity will be blocked until they respond to the alert dialog. A listener function is used to handle the user's button press in the alert dialog. We will use an alert in our next project example, but for now, here is the API information:

**`native.showAlert( title, message [, {buttonLabels} [, listener]] )`**

Thus, a showAlert like:

```
native.showAlert("Mom called", "You are in SO MUCH TROUBLE!", {"Hide Evidence", "I Didn't Do It"})
```

Results in an alert like:



## GPS

If you have ever tried to find your way through an area that you are not familiar with, then you know how wonderful applications that help you navigate using GPS services can be. Having recently tried to find my way around near downtown Boston, I can attest to how wonderful these services are to the weary traveler. Need a coffee, restaurant, or gas station? Location aware services make it possible to find the closest vendor and how to get there. Location or GPS events are generated by the GPS hardware within the smart phone with the data being supplied to the runtime object (just like for the Accelerometer and gyroscope). GPS services can supply longitude, latitude, altitude, rate of movement, and direction. You have control over the accuracy of the GPS event.

The GPS location event is based off of three possible location services. The most accurate is off of GPS satellites that orbit the Earth (as opposed to Mars, which wouldn't be much help). Through triangulation, your phone is able to determine your location. The second method, used more frequently, is through triangulation using cell towers, which take in to account the amount of time it took the signal to get to the tower, the angle of the signal, etc. The final method is by using wireless routers. This is the least accurate method. Your smartphone actually uses a combination of these methods to determine location. This is known as Assisted-GPS.

More than any other runtime system, GPS can be a real battery drain. Generally, the higher the accuracy, the more drain. You should also ask your users if location services can be used with the app. Most app stores require the users approval before their location can be used in your app.

So let's make an app to figure out where we are at! If the app is going to be used on an Android device, you need to include a few permissions in the build.settings file.

## Project 12.2 GPS

```
build.settings
settings =
{
    android =
    {
```

```

usesPermissions =
{
    -- Permission to access the GPS.
    "android.permission.ACCESS_FINE_LOCATION",

    -- Permission to retrieve current location from WiFi
or cellular service.
    "android.permission.ACCESS_COARSE_LOCATION",
},
}

```

Now on to the main lua file. I am again using a simplified version of the app supplied in the sample folder. In this particular app, we will learn our longitude, latitude, speed, direction, and altitude. If it is run in a simulator, it will return a pre-defined location. If you look at the background file included, you will see that the text describing each of the fields is already included. All we need to do is provide the GPS data.



main.lua

```

local currentLatitude = 0
local currentLongitude = 0

display.setStatusBar( display.HiddenStatusBar )

local background = display.newImage("gps_background.png")

local latitude = display.newText( "--", 0, 0, nil, 26 )
latitude.anchorX=0
latitude.anchorY=.5
latitude.x, latitude.y = 125 + latitude.contentWidth * 0.5, 64
latitude:setFillColor(1,.3,.3)

```

```

local longitude = display.newText( "--", 0, 0, nil, 26 )
longitude.anchorX=0
longitude.anchorY=.5
longitude.x, longitude.y = 125 + longitude.contentWidth * 0.5,
latitude.y + 50
longitude:setFillColor(1,.3,.3)

local altitude = display.newText( "--", 0, 0, nil, 26 )
altitude.anchorX=0
altitude.anchorY=.5
altitude.x, altitude.y = 125 + altitude.contentWidth * 0.5,
longitude.y + 50
altitude:setFillColor(1,.3,.3)

local accuracy = display.newText( "--", 0, 0, nil, 26 )
accuracy.anchorX=0
accuracy.anchorY=.5
accuracy.x, accuracy.y = 125 + altitude.contentWidth * 0.5,
altitude.y + 50
accuracy:setFillColor(1,.3,.3)

local speed = display.newText( "--", 0, 0, nil, 26 )
speed.anchorX=0
speed.anchorY=.5
speed.x, speed.y = 125 + speed.contentWidth * 0.5, accuracy.y +
50
speed:setFillColor(1,.3,.3)

local direction = display.newText( "--", 0, 0, nil, 26 )
direction.anchorX=0
direction.anchorY=.5
direction.x, direction.y = 125 + direction.contentWidth * 0.5,
speed.y + 50
direction:setFillColor(1,.3,.3)

local time = display.newText( "--", 0, 0, nil, 26 )
time.anchorX=0
time.anchorY=.5
time.x, time.y = 125 + time.contentWidth * 0.5, direction.y + 50
time:setFillColor(1,.3,.3)

```

Now that we have the variables setup, we can check for potential problems. In the function below, we will use an alert to notify the user of potential problems. If there are no problems, the response from the GPS system will fill in the remaining data.

```

local locationHandler = function( event )

    -- Check for error (user may have turned off Location
Services)
    if event.errorCode then
        native.showAlert( "GPS Location Error",
event.errorMessage, {"OK"} )
        print( "Location error: " .. tostring(
event.errorMessage ) )
    else

        local latitudeText = string.format( '%.4f',
event.latitude )
        currentLatitude = latitudeText
        latitude.text = latitudeText
        latitude.x, latitude.y = 125 + latitude.contentWidth *
0.5, 64

        local longitudeText = string.format( '%.4f',
event.longitude )
        currentLongitude = longitudeText
        longitude.text = longitudeText
        longitude.x, longitude.y = 125 + longitude.contentWidth *
0.5, latitude.y + 50

        local altitudeText = string.format( '%.3f',
event.altitude )
        altitude.text = altitudeText
        altitude.x, altitude.y = 125 + altitude.contentWidth *
0.5, longitude.y + 50

        local accuracyText = string.format( '%.3f',
event.accuracy )
        accuracy.text = accuracyText
        accuracy.x, accuracy.y = 125 + accuracy.contentWidth *
0.5, altitude.y + 50

        local speedText = string.format( '%.3f', event.speed )
        speed.text = speedText
        speed.x, speed.y = 125 + speed.contentWidth * 0.5,
accuracy.y + 50

        local directionText = string.format( '%.3f',
event.direction )
        direction.text = directionText

```

```

        direction.x, direction.y = 125 + direction.contentWidth
* 0.5, speed.y + 50

        -- Note: event.time is a Unix-style timestamp, expressed
in seconds since Jan. 1, 1970
        local timeText = string.format( '%.0f', event.time )
        time.text = timeText
        time.x, time.y = 125 + time.contentWidth * 0.5,
direction.y + 50
    end
end

-- Determine if running on Corona Simulator

local isSimulator = "simulator" == system.getInfo("environment")

-- Location Events is not supported on Simulator

if isSimulator then
    msg = display.newText( "Location events not supported on
Simulator!", 0, 230, "Verdana-Bold", 13 )
    msg.x = display.contentWidth/2      -- center title
    msg:setFillColor( 1, 1, 1 )
end

```

And now we can call the GPS/location system as a Runtime event.

```

-- Activate location listener
Runtime:addEventListener( "location", locationHandler )

```

As you can see, it is easy to make get the data from the smartphone with Corona. The challenge is using the data effectively!

## Maps

Having data is one thing. Turning it into information is another. By using GPS with a map, you can easily determine your location and how to get to where you want to be, or determine what is available in your local area. Corona includes 3 sets of tools for working with maps: the Map object, mapAddress events, and mapLocation events. Between these three API objects, you can create some neat tools.

**Note:** Map objects and the associated API calls will not work in the current Corona Simulator. You will need to build the app and either run it in the Apple xCode simulator or deploy to a device.

`native.newMapView(left, top, width, height)` – like `display.newImage` or `display newText`, `native.newMapView()` creates the map object.

### Map Object

- `object:addMarker(latitude, longitude [, options])` – sets a marker at the provided location. A title and subtitle can be included in the options table.
- `object:getUserLocation()` – returns users current GPS location in a table.
- `object.isLocationVisible` – returns a Boolean if the user's location is currently visible on the map.
- `object.isScrollEnabled` – allows the user to scroll the map by hand. Set to false if you want to lock the map location.
- `object.isZoomEnabled` – allows user to use pinch/zoom gestures on the map.
- `object.mapType(map string)` – sets the map to standard (default) , satellite, or hybrid mode.
- `object:nearestAddress(latitude, longitude [, handler])` – returns nearest address on given latitude & longitude. Address is returned as a Map Address event (see below).
- `object:removeAllMarkers()` – As the name implies, removes pins from the map.
- `object:requestLocation(address string)` – returns the longitude & latitude of an address.
- `object:setCenter(latitude, longitude [, isAnimated])` – moves center of map to the given latitude & longitude.
- `object:setRegion(latitude, longitude, latitudeSpan, longitudeSpan [, isAnimated])` – moves map region to new location with center point and horizontal and vertical span (i.e. how much is shown on the screen, or level of zoom) given in degrees of latitude and longitude.

**Map Address** is created by the `object:nearestAddress()` and returns the following event fields:

- `event.city`
- `event.cityDetail`
- `event.country`
- `event.countryCode`
- `event.errorMessage`
- `event.isError`
- `event.name`
- `event.postalCode`
- `event.region`
- `event.regionDetail`

- event.street
- event.streetDetail

**Map Location** is created by object:requestLocation() and returns the following event fields:

- event.errorMessage
- event.isError
- event.latitude
- event.longitude
- event.name

## Project 12.3 Maps

Let's try out the map code for ourselves. Remember, this code will generate an error message in the Corona Simulator, so you will have to build and deploy to either the xCode simulator (which will then center on Apple Headquarters), or to a device.

So that the app will work on Android devices, you will need the following permissions in the build.settings file.

build.settings

```
settings =
{
    android =
    {
        usesPermissions =
        {
            "android.permission.INTERNET",
            -- Permission to access the GPS.
            "android.permission.ACCESS_FINE_LOCATION",

            -- Permission to retrieve current location from WiFi
            or cellular service.
            "android.permission.ACCESS_COARSE_LOCATION",
        },
    },
}
```

Like most Corona API tools, using the map feature is very straightforward. Once the new view is created, we can set the type of map, and its location on the screen. We can also set the maps center, retrieve latitude and longitude based upon an address, and then set a marker (or pin) at that location.

```

main.lua
local myMap = native.newMapView(0, 0, display.contentWidth,
display.contentHeight)

myMap.mapType = "hybrid"

myMap.x = display.contentWidth/2
myMap.y = display.contentHeight/2

-- center on Apple's headquarters to begin
myMap:setCenter( 37.331692, -122.030456 )

local latitude, longitude = mymap:requestLocation( "1900
Embarcadero Rd., Palo Alto, CA 94303" )

myMap:addMarker(latitude,longitude,{title="Corona Labs"})
myMap:setCenter( latitude, longitude )

```

## Summary

The system tools open up all kinds of possibilities for apps and games (I have at least 5 augmented reality games currently on my personal ‘apps to make’ list)! I am sure that you see many opportunities for how the smartphones built-in features can make your app even more useful.

## Assignments

1. Create an app that uses the accelerometer to move an object around on the screen.  
Note: this is the feature many apps use to manage rolling a ball through a maze.
2. Create an app based upon the map feature and add a pin for your home address.
3. Add an alert to an app that you have previously made.
4. Using Maps and Location services make an app that shows the map of your current location.
5. Challenge: Create an app that uses the gyroscope feature to keep a round object from rolling off a static beam or line. You will need to use physics and play with the gravity settings for this one!

# Chapter 13: File Input/Output

## Learning Objectives

I know, I know, you have been wanting to ask the following question for some time now, “How do I save information in my app?” With a few great API resources, we can take care of this problem easily. In this chapter we will learn:

- Where to store files on the smartphones and tablets
- The difference between explicit and implicit file input and output
- How to read data from a file
- How to write data to a file
- Using JSON to store and retrieve app data

## File I/O Storage Considerations

While we have briefly discussed possible file locations previously, I think it would be a good idea to review. There are three file locations 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 app directory.
- system.ResourceDirectory – is the folder or directory where your assets are stored. Never change anything in this folder! 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 while the app is active, 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, databases, etc) are stored in a

location that is off-limits to any other application that is installed. All of your files will be located in a specific directory for your app.

## Reading Data

I am sure that you have noticed that many apps and games take a few seconds (or even minutes) to load. This is primarily due to needing to load a large quantity of data or graphics into memory. By storing data in external files we are able to accomplish several important features.

If security is a concern, either because we are using personal data from those who use our app or the need to safe guard proprietary information, we can encrypt data files making it much more difficult to hack or steal the information. If the data is kept un-encrypted, it can be accessed by others.

If I have created a complex game, I will usually need to load a lot of information about the level. While I can program all of the level information into the game, using that approach makes it much more difficult to provide updates, bug-fixes, or new levels to the game without resubmitting the app for app store approval. By storing as much information as possible in external files, I can download updates to my users without needing to go through the approval process again (we will discuss how to download files for you app in chapter 15).

## Implicit vs. Explicit Files

Lua (and by extension, Corona) has two different types of file input and output: 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). While at first consideration it might seem strange to think of the Corona Terminal as a 'file', it is by traditional Computer Science considerations. In the 'old' days of computer usage (i.e. pre-1980's), any operation that wrote or read information to or from a location was file input or output (file I/O). In the early days of massive computer systems, to be able to output information to a terminal was very similar to writing data to a file or sending the information to a printer. Thus, over time as programming languages have developed, reading or writing to the Corona Terminal window is still considered a file I/O operation. Consider it a long winded way of doing a print command.

**Explicit** file operations allow the reading and writing of typical (i.e. not Corona Terminal) files including text files and binary files. For the majority of your file operations you will be using the explicit file API tools. 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(filehandle)` – 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, temporary 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] )` – sets 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.

## Project 13.0 Reading & Writing to a File

In the following examples we will create simple apps to write and read data to a text file that will be stored in the documents folder of the smartphone or tablet. We will use both implicit and explicit API calls to accomplish our write and read.

One of the first decisions you must make when preparing to write a file is if you need to preserve information that was previously written to the file. When opening the file to write, you must decide if you are over-writing/erasing all previous information (“w”), or appending to previously written data (“a”). When opening a file for writing or appending, if the file does not exist, it will be created.

To keep things simple, I am going to assume that I am not concerned about previously saved files. So, I will use the “w” mode to create or overwrite any previous file.

`main.lua`

```
-- set the path to the documents directory
pathDest = system.pathForFile( "ch13Write",
system.DocumentsDirectory )

-- open/create the file
local myFile = io.open( pathDest, "w" )

myFile:write("Hi Mom! I made a file")
```

```
myFile:flush()  
io.close(myFile)
```

When you are finished writing information to the file, you should always flush the data. This will ensure that everything has been written to the file before it is closed.

Now we will add the code to read the data from the file and display it to the screen. After verifying that the file exists, we can load all of the file's contents by using the “\*a” parameter.

```
-- check that the file was created  
myFile = io.open( pathDest, "r" )  
if myFile then  
    -- the file exists, read the data  
    local contents = myFile:read( "*a" )  
  
    local myOutput ="Contents of \n" .. pathDest .. "\n" ..  
contents  
    io.close(myFile)  
    display.newText(myOutput, 50, 50, nil, 16)  
end
```

Now that we have successfully created a file, let's look at an example of a file that can be appended.

### Project 13.1 Appending & Reading from a File

Building on project 13.0, we will now use the append command. The advantage of this command is data will be added each time we run the program. Thus, if we run the app multiple times, it will continue to add new lines of data. To start using append, we only need replace the “w” with “a” in the initial open command:

```
-- set the path to the documents directory  
pathDest = system.pathForFile( "ch13Write",  
system.DocumentsDirectory )  
  
-- open/create the file  
local myFile = io.open( pathDest, "a" )  
  
myFile:write("Hi Dad! I made a file \n")  
myFile:flush()  
io.close(myFile)
```

```

-- check that the file was created
myFile = io.open( pathDest, "r" )
if myFile then
    -- the file exists, read the data
    local contents = myFile:read( "*a" )
    local myOutput ="Contents of \n" .. pathDest .. "\n" ..
contents
    io.close(myFile)
    display.newText(myOutput, 50, 50, nil, 16)
end

```

By adding the “\n” to the end of our write command, we can easily see the additional lines of text since \n forces a new line to the display.

Run this app a few times to see the impact of append.

## JSON

You might have heard of JSON (pronounced Jason) before and wondered, “Who is this guy Jason, and what does he have to do with my computer?” JSON stands for JavaScript Object Notation and is a popular method of encoding information for storage. It is considered a lightweight alternative to XML (eXtensible Markup Language) and is fully integrated and supported by Corona. JSON data can be very detailed with tables stored within the data.

Now I know what you are thinking: “That’s nice, so what?” (scary how I can do that, isn’t it?)

Consider this problem: I have been working hard on a mobile game or app and I need to organize the information that will be stored in an easy to read format. In other words, JSON is easy to read for the computer and people. Data that has been formatted into a JSON format would look like this:

```
{
    ["First Name"] = "Brian",
    ["Last Name"] = "Burton",
    ["Level"] = 9,
```

```

        ["Score"] = 434434,
        ["Location"] = {4.8, 15.16, 23.42},
        ["Avatar"] = "Blue42"
    }

```

Now I have information that can be easily passed to the computer app, but is still in a format that most people can easily understand.

Basic JSON commands include decode, encode, and null. JSON is an external library, so it does need to be loaded with 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).

Note that we are using tables when working with JSON! What is encoded and decoded is in the format of an array table. Thus, when I decode a JSON string, it will be in the format of a table with the key being whatever I set (such as “First Name”).

## Project 13.2 File I/O with JSON

So let’s do a simple app to demonstrate how easy it is to use JSON. We will do a more complex application when we get to networking (Chapter 15). We will first require JSON and setup the data. Note that while I am using strings and numbers to pass to the JSON array table, I could just as easily pass variables.

```

local json = require "json"

-- JSON script:

local data = {
    ["First Name"] = "Brian",
    ["Last Name"] = "Burton",
    ["Level"] = 9,
    ["Score"] = 434434,
    ["Location"] = {4.8, 15.16, 23.42},
    ["Avatar"] = "Blue42.png"
}

```

Once we have our data loaded, we can encode it easily with the json.encode command. To give you an idea of what it looks like encoded, we will do a print command to show the encoded data.

```
local jsonBlob = json.encode (data)
print (jsonBlob)
```

Next we will setup our file so that the data can be saved.

```
-- set the path to the documents directory
pathDest = system.pathForFile( "ch13JSON",
system.DocumentsDirectory )

-- open/create the file

local myFile = io.open( pathDest, "w" )

myFile:write(jsonBlob)
myFile:flush()
io.close(myFile)

-- check that the file was created
myFile = io.open( pathDest, "r" )
if myFile then
    -- the file exists, read the data
    local contents = myFile:read( "*a" )

    io.close(myFile)
```

And now we will decode our data. To show how to extract the data from the array table, I have included two display commands to output the data.

```
local myOutput = json.decode(contents)

display.newText(myOutput["First Name"], 50, 50, nil, 16)
display.newText(myOutput["Level"], 50, 100, nil, 16)
end
```

## **Summary**

File input and output gives us the ability to create apps with persistent data. If you have ever had to enter the same data more than once into a mobile application or game, you know how nice data persistence is in an app! We also learned about using JSON to format the data in a user friendly method.

## **Assignments**

1. Create an app that will allow the user to enter their name and age and store & read the data in an external file.
2. Create an app that will allow the user to enter their name and years in school. The data should be stored in a JSON format.
3. Building on Assignment 1, build an app that checks to see if the file already exists, if it does, welcome the user back to the app and give them a button that will allow them to “logout” and enter new information. Save the new information to the file.
4. Building on Assignment 2, build an app that checks to see if the file already exists, if it does, welcome the user back to the app and give them a button that will allow them to “logout” and enter new information, storing the new information in a JSON format.

# Chapter 14 Working with Databases

## Learning Objectives

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:

- What a Database is and why you would want to use one
- How Databases are structured
- Creating a database for your app
- Reading from a SQLite database
- Writing to a SQLite database

## Database Defined

A database in its simplest form is a way to organize a collection of data. Usually the data being stored is related in some fashion. For example, if I wanted to organize information about students in one of my classes, I could create a database that contained their identification information, how they did on assignments, where they sit, who their best friend is, etc. And just so you know, the word data is plural. When you have only one piece of data, you would use the singular, datum... unless you are talking about the Star Trek character, then it is Data.

## Database Software

There are many database programs available to help you create a database. At the personal use end of the spectrum, we have Microsoft Access. For small to mid-size businesses commonly used databases include MySQL and PostgreSQL to databases for huge projects that use Oracle and IBM DB2, and Sybase.

For mobile devices, the database most commonly used is SQLite. Apple began the trend by incorporating SQLite into the early iPhones. Now it is used by most every smartphone or tablet. For more information on SQLite I recommend their website: <http://www.sqlite.org>.

Most databases utilize SQL (Structured Query Language) to handle storing or retrieving information. The basics of SQL are easy to learn (we will cover a few commands in this and later chapters), but can become very complex. There are minor variations of SQL depending on the database, but the basics are fairly consistent across the various databases.

## How a Database is Structured

As the purpose of a database is to organize data into a structure, it is a good idea to become familiar with the basic structure of a database. Let me emphasize that this is a simplified presentation of how databases are structured. Let's start at the most basic component and work up.

The simplest part of a database is the field. A field holds one type of data such as an ID number, a name (first, last, or combined), a phone number, etc. When you are creating a field, you specify the type of data that will be stored in the field: String, integer, decimal, Boolean, blob, binary, etc.

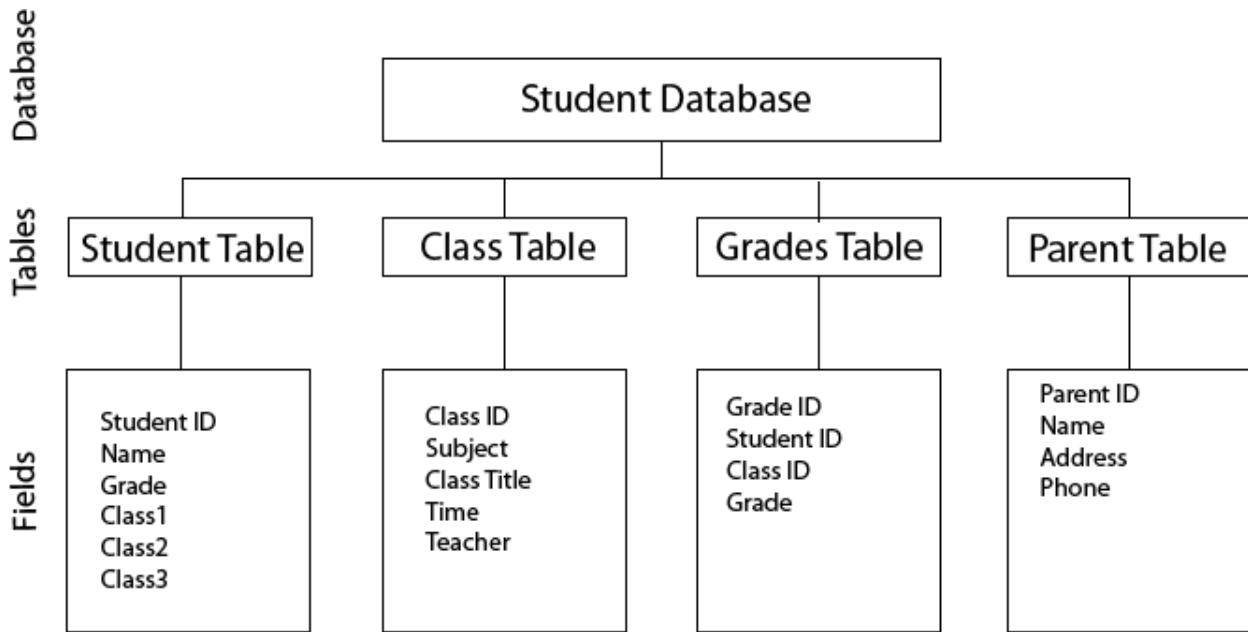
When I complete all the fields on a specific subject, it is called a record. In the example below, I can create a record of a student. The student record will have the student's ID, name, their current grade level, and the classes that they are enrolled.

A collection of fields that hold related information are organized into a table. In the example below I have a student table, a class table, a grades table, and a parent table. Some people like to think of the information that is created to this point as a spread sheet, where the columns would be fields of information, records would be rows, and the spreadsheet would be the table.

	A	B	C	D	E	F
1	ID	Name	Grade	Class1	Class2	Class3
2		1 Beth	9	312	354	310
3		2 John	10	411	422	310
4		3 Cindy	10	422	454	312

Unfortunately the spreadsheet analogy quickly breaks down as we begin to add additional tables and more complex databases. But for the time being, if it helps you to think of your data organized as a spreadsheet, by all means, do so!

The container for all of our tables is the database itself. A database pulls all of this now organized data and stores it efficiently as a single file. A database will also usually contain one or more indexes to help locate the data in a fast and efficient manner.



As I mentioned, this is a quick introduction into databases and how they are organized so that you can create apps that use them. If you want to go deeper into databases, you will most certainly have a very lucrative career working with big data and helping organizations manage the information explosion that is occurring today!

## How to create a Database

Let's look at how to open or create a database for mobile devices on our PC or Mac, then we will use an existing database for our first app. One of the advantages of SQLite is that it is public domain, so freely available to anyone who wishes to use it. There are several great tools available for viewing, creating, or modifying SQLite databases. The one that I use and will demonstrate is a plugin for Firefox. If you would like to use something else, just do a search for SQLite manager. The Firefox plugin can be downloaded from <https://code.google.com/p/sqlite-manager/> or just do a search in Tools > Add-ons of your favorite browser.

To create a SQLite database with the SQLite Manager, click either File > New Database or the New Database icon. You will get a popup window. Enter a name for your database.



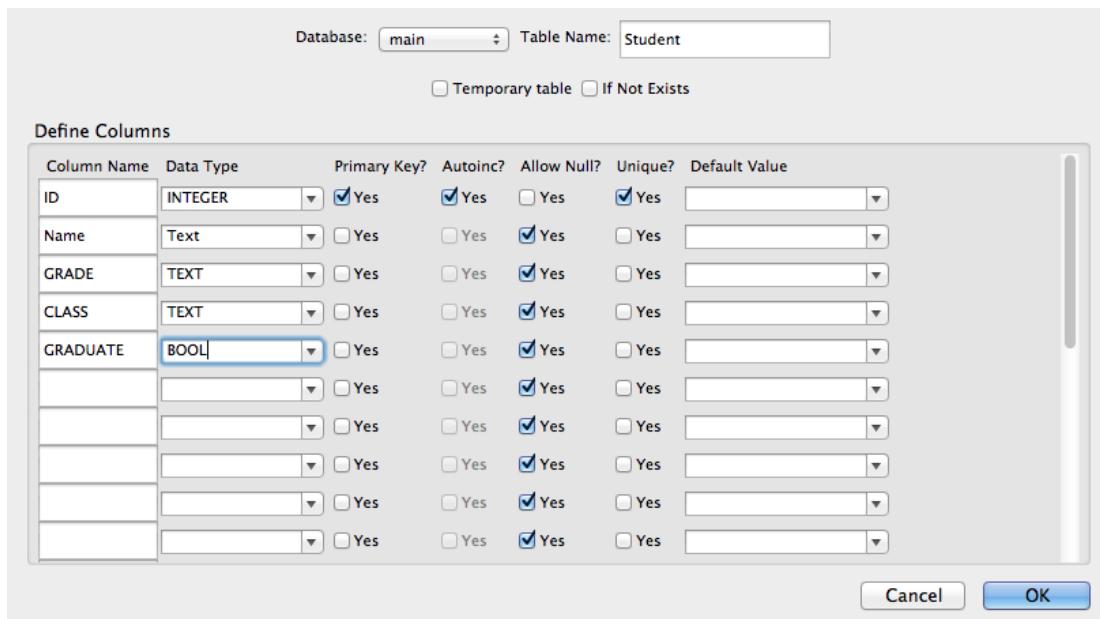
Once you have the database named, it is time to create a table. You can create a new table for the database by selecting Table > Create Table from the menu bar or clicking on the Create Table icon. You will need a name for the table, preferably different from the name of the database and your potential field names.

Time to define your table. Going back to our spreadsheet analogy, think about the names for the different columns (fields) that will be in your table. The first column should always be used for a primary key, which will be used to help keep your data organized. I used the name ID for my primary key. You will next set the data type that will be stored in the field/column. You need to think about the type of information that will be stored in the field. Will it only be numbers? Text (including information such as a phone number or ID number)? Or is the information limited to true or false and should be set to a Boolean?

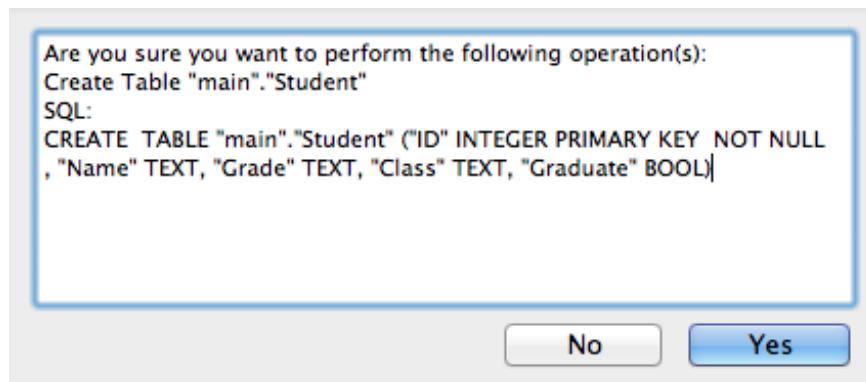
For the ID field, since it will only be used to keep track of the information entered in this table as an index, I set the data type to Integer. I then checked the Primary Key checkbox so that the database knows this will be used to organize the information in this table. If you would like SQLite to keep track of the Primary Key (which is recommended), you should also check Autoinc, which is auto-increment – meaning it will automatically fill in the next larger integer in the ID for you each time a new record is created.

For the remaining fields of the example table, I created a basic student profile that a school or college might use for tracking. I set each field data type to text except for Graduate, which could be set as a Boolean since it is either true or false.

Once you have create your table, click okay. You are now ready to enter information into your database.

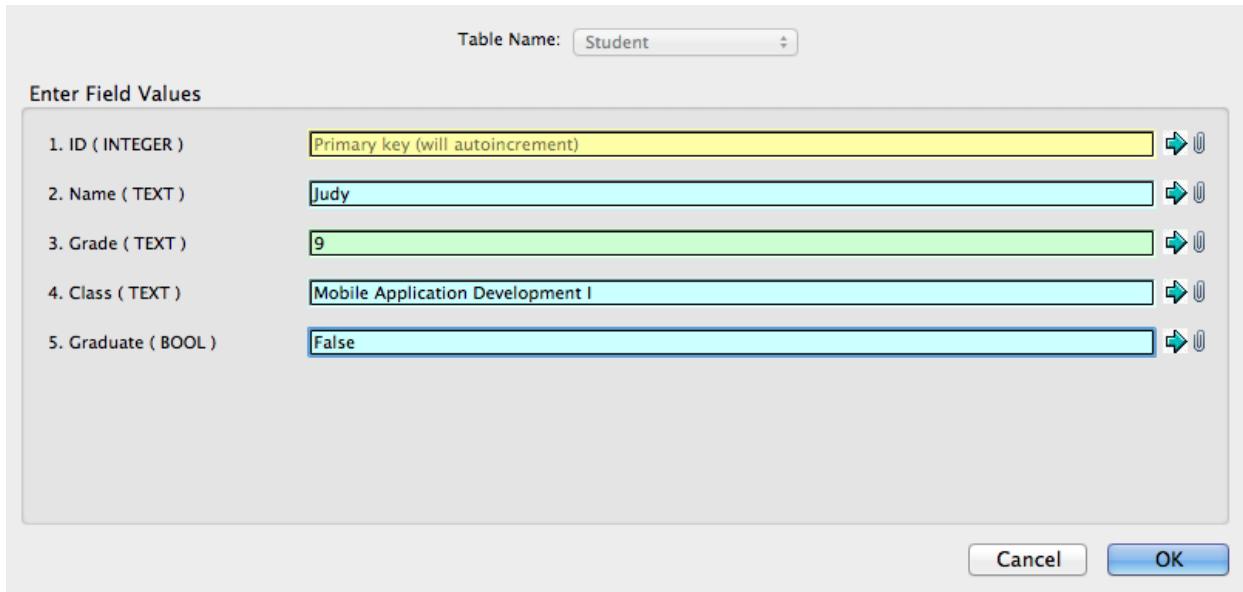


When you click okay, you will be asked to confirm the table creation.

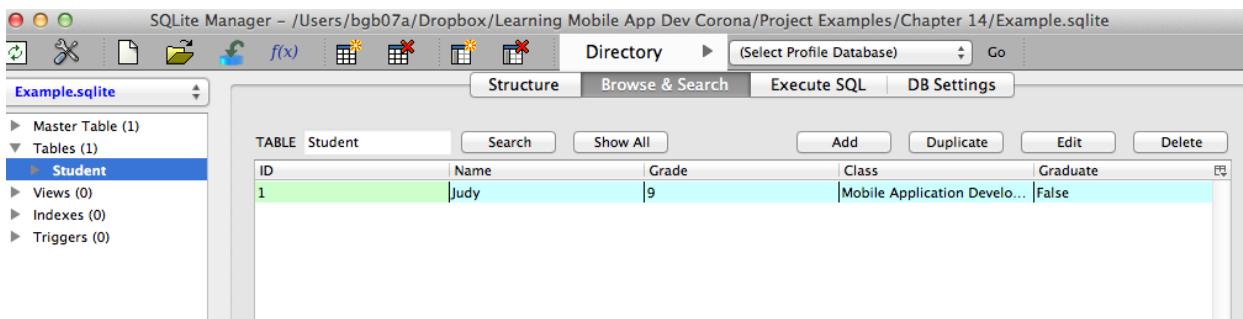


After the table is created, you will be able to see the table in the browser. Of course, it doesn't have any data yet, so it will only show the column or field headings.

Clicking the Add button will open the Add Record window. Go ahead and create a new record, then click OK.



After the record is added, you will see it listed in your Browse and Search listing.



That is a quick and dirty introduction into how to create a SQLite database using SQLite Manager. While you may not have a full grasp of what is possible with a database, it should give enough of an understanding to be able to create mobile apps that use pre-existing databases or how to store information to a database.

## Working with a Database

Corona includes native SQLite support for iOS and compiled version (adding a mere 300K to your app size) for Android. We are going to continue to keep things simple and use just the basic commands for accessing a local database that is stored on the local smart phone or tablet.

The basic SQLite commands are:

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.

fileVariable:exec (SQL Command) – executes a SQL command in the database. Typically used to create tables, insert, update, append or retrieve data from a database.

fileVariable:nrows(SQL Command) – returns successive rows from the SQL statement.

fileVariable:close() – closes the database.

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>.

## LuaSQLite Commands

Luasqlite3 is an external library, thus requires the use of

```
local sqlite3 = require "sqlite3"
```

prior to any SQLite calls.

## Project 14.0: 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 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"  
    }  
}
```

In our main.lua, we will first 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. It is recommended that you do not read from (and never write to) a database stored in the resource directory. Doing so could flag your app as a virus or malware by the operating system! To resolve this 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 to read from the resource folder and 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 persistence)  
local path = system.pathForFile("zip.sqlite",  
system.DocumentsDirectory )
```

We now have the variable path set to the location of zip.sqlite (if it exists) in the documents directory. In the next few lines, if there isn't a copy of zip.sqlite in the documents directory,

we will use a couple of the file IO commands we have learned previously to copy the file from the resource directory to the documents directory.

Oh, and to make sure it works on the Windows simulator, be sure to use “rb” and “wb” when copying the file. This forces the Windows simulator to copy the file as a binary instead of a text file. Otherwise you will get a “Malformed Database File” error (yes, I learned that one the hard way).

```
file = io.open( path, "r" )
if( file == nil )then
    -- Database doesn't already exist, so copy it from the
    --resource directory
    pathSource = system.pathForFile( "zip.sqlite",
system.ResourceDirectory )
    fileSource = io.open( pathSource, "rb" )
    contentsSource = fileSource:read( "*a" )
    --Write Destination File in Documents Directory
    pathDest = system.pathForFile( "zip.sqlite",
system.DocumentsDirectory )
    fileDest = io.open( pathDest, "wb" )
    fileDest:write( contentsSource )
    -- Done
    io.close( fileSource )
    io.close( fileDest )
end

-- One way or another the database exists
-- So open database connection

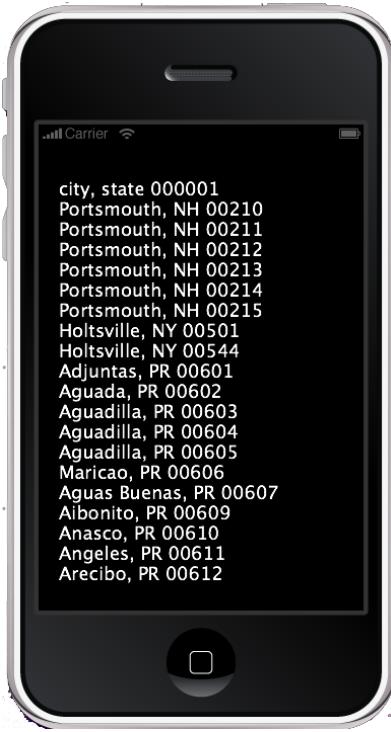
db = sqlite3.open( path )
```

Next, we will setup some code to handle an applicationExit event, so that the database will be properly closed should the user hit the home button or the phone rings, or something else unexpected happens.

```
-- 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, we will use a SQL select statement. SELECT tells the database to return all the rows that meet our search criteria. Since I am using \*, which means wildcard, or "Give me everything!" that is in the zipcode table, I am going to limit the number of rows returned to the first 20. We will use a for loop to display the content of those rows to my device display:

```
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:setFillColor(1, 1, 1)
end
```

And finally, we setup the system event listener for the close event that was handled earlier.

```
-- system listener for applicationExit
Runtime:addEventListener ("system", onSystemEvent)
```

And that is how we make an app to read from a database that already exists. Next, let's save some information to a database.

## Project 14.1 Writing to a SQLite Database

In this project, we are going to do one of the most requested types of apps: an app that can store information to a local database and retrieve it for later use. We will use 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 that is similar to the one we created earlier in the chapter. 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. We are going to use composer to move back and forth between the screens.

To simplify 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 Composer, the main Lua file primarily serves as a starting place for your app. We begin by loading the Composer and calling the next scene.

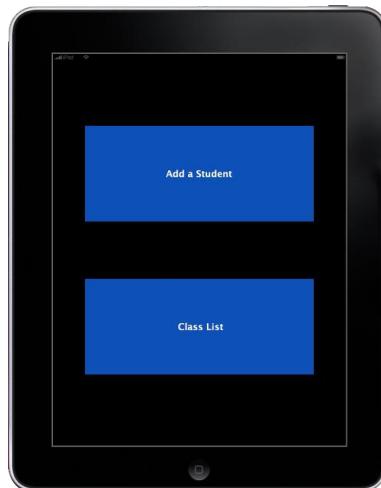
```

local composer = require("composer")
composer.gotoScene ("menu")

```

## 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 handles the require for widgets (for our buttons) and setting up everything for Composer.



After setting our variables, we can then create widget based buttons and functions to go to the correct scene.

```

local widget = require ( "widget" )
local composer = require("composer")
local scene = composer.newScene()
local localGroup = display.newGroup()

```

```

--Called if the scene hasn't been previously seen
function scene:create ( event )
    local displayClass_function = function ( event )
        composer.gotoScene( "displayClass", "fade", 400 )
    end

    local displayClass_button = widget.newButton{
        defaultFile = "menuButton.png",
        label = " Class List",
        size = 24,
        emboss=true,
        onRelease = displayClass_function,
        id = "displayClass"
    }

    local addStudent_function = function ( event )
        composer.gotoScene( "addStudent", "fade", 400 )
    end

    local addStudent_button = widget.newButton{
        defaultFile = "menuButton.png",
        label = "Add a Student",
        size = 24,
        emboss=true,
        onRelease = 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

    localGroup:insert(displayClass_button)
    localGroup:insert(addStudent_button)
end

function scene:show(event)
    localGroup.alpha = 1
end

function scene:hide(event)
    localGroup.alpha = 0
end

-- "create" is called whenever the scene is FIRST called
scene:addEventListener( "create", scene )

-- "show" event is dispatched whenever scene transition has finished

```

```

scene:addEventListener( "show", scene )

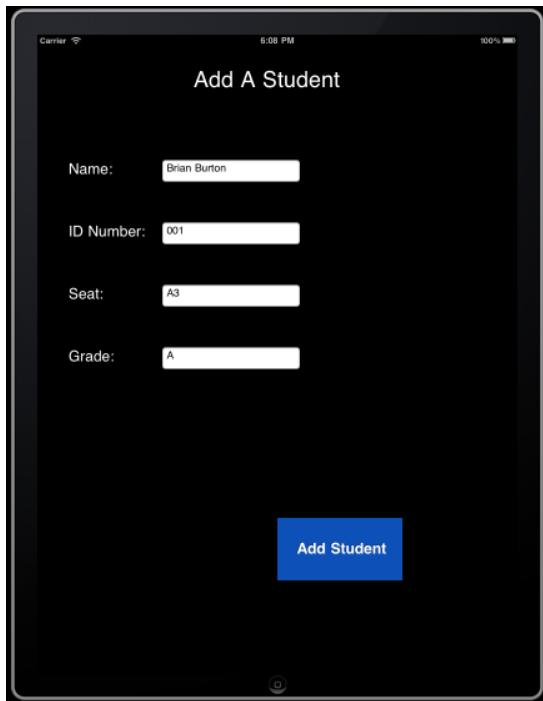
-- "hide" event is dispatched before next scene's transition begins
scene:addEventListener( "hide", scene )

return scene

```

### **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. Adding a few comments on the purpose program at the beginning of each file is good programming practice. There is nothing worse than trying to figure your program logic six months later when the program needs revised or updated.



We first handle the requires and setup the scene and new group needed in addStudent.lua file. To get started, we will setup the labels and textboxes as a part of our createScene function .

```
=====
-- SCENE: addStudent
=====
```

```

--[ [
- INFORMATION
- add a student to the SQLite database.
--]]]

local widget = require ( "widget" )
local composer = require("composer")
local scene = composer.newScene()
local localGroup = display.newGroup()
--include SQLite
    require "sqlite3"

--Called if the scene hasn't been previously seen
function scene:create ( event )

    -- Add textboxes and labels for data entry
    local title = display.newText(localGroup, "Add A Student", 250,
50, native.systemFont, 36)
    title:setFillColor(1, 1, 1)
    local nameLabel = display.newText(localGroup, "Name:", 50, 200,
native.systemFont, 24)
    nameLabel:setFillColor(1, 1, 1)
    local idLabel = display.newText(localGroup, "ID Number:", 50, 300,
native.systemFont, 24)
    idLabel:setFillColor(1, 1, 1)
    local seatLabel = display.newText(localGroup, "Seat:", 50, 400,
native.systemFont, 24)
    seatLabel:setFillColor(1, 1, 1)
    local gradeLabel = display.newText(localGroup, "Grade:", 50, 500,
native.systemFont, 24)

    local studentName = native.newTextField(200, 200, 220, 36)
    studentName.inputType="default"
    localGroup:insert(studentName)

    local studentID = native.newTextField(200, 300, 220, 36)
    studentID.inputType="default"
    localGroup:insert(studentID)

    local seat = native.newTextField(200, 400, 220, 36)
    seat.inputType="default"
    localGroup:insert(seat)

```

```
local studentGrade = native.newTextField(200, 500, 220, 36)
studentGrade.inputType="default"
localGroup:insert(studentGrade)
```

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 device's documents directory. If the file does not already exist, it is created on the first call. By storing the database in 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 submit student data
local submitStudent_function = function ( event )

    -- open SQLite database, if it doesn't exist, create database
    local path = system.pathForFile("students.sqlite",
system.DocumentsDirectory)
    db = sqlite3.open( path )
    print(path)
```

Once the file is open, we need to set up 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)
```

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 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 concatenation 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.

```

-- save student data to database
local tablefill ="INSERT INTO myclass VALUES (NULL,'" ..
studentName.text .. "','" .. studentID.text .. "','" .. seat.text ..
"','" .. studentGrade.text ..');");
print(tablefill)
db:exec( tablefill )

```

After we have executed the SQL insert statement, the database is closed.

```

-- close database
db:close()
print("db closed")

```

Now that we have written the data and closed the database, we can prepare the scene to return to menu.lua. First we must remove the 4 textfields using the removeSelf method (otherwise they will remain on the screen since they are not part of the OpenGL canvas). Then we can call the composer.gotoScene to return to the menu.

```

-- Clear textFields & return to menu screen
studentName:removeSelf()
studentID:removeSelf()
seat:removeSelf()
studentGrade:removeSelf()
composer.gotoScene( "menu", "fade", 400 )
end -- submitStudent_function

```

Next we create the addStudent button.

```

local addStudent_button = widget.newButton{
    defaultFile = "selectButton.png",
    overFile = "selectButton.png",
    label = " Add Student",
    size = 24,
    emboss=true,
    onRelease = submitStudent_function,
    id = "addStudent"
}
addStudent_button.x = display.contentWidth/2+100
addStudent_button.y = display.contentHeight-200

-- 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)
end -- scene:create function
```

With the createScene function finished, we can take care of the other functions needed for composer to work properly. Next we will take care of the show, which is called when the scene transition has finished.

```
function scene:show(event)
    localGroup.alpha=1
end
```

That just leaves handling what to do when the scene exits and setting up the scene listeners.

```
function scene:hide(event)
    localGroup.alpha = 0
end

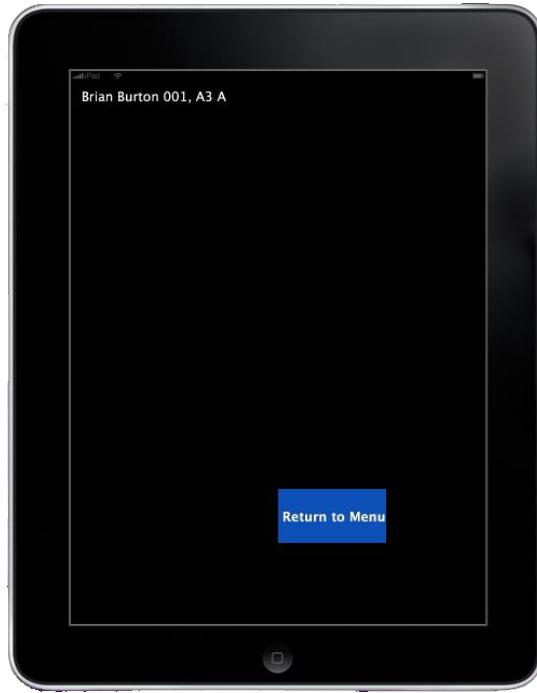
-- "create" is called whenever the scene is FIRST called
scene:addEventListener( "create", scene )

-- "show" event is dispatched whenever scene transition has finished
scene:addEventListener( "show", scene )

-- "hide" event is dispatched before next scene's transition begins
scene:addEventListener( "hide", scene )
return scene
```

## displayClass.lua

The displayClass.lua file handles the displaying of all student data to the device screen. We will keep this scene very simple, just displaying the information from the database as a screen text object, though we could just as easily show it as a Table View or in some other format.



```
=====
-- SCENE: displayStudent
=====

-- [ [
*****
-- INFORMATION
*****
- Display class information to screen.
-- ] ]

local widget = require ( "widget" )
local composer = require("composer")
local scene = composer.newScene()
local localGroup = display.newGroup()
--include SQLite
    require "sqlite3"
```

Next we will set up the createScene which opens the database.

```
--Called if the scene hasn't been previously seen
function scene:create ( event )

    -- 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:setFillColor(1, 1, 1)
    localGroup:insert(t)
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 )
    -- return to menu screen
    composer.gotoScene( "menu", "slideRight", 400 )
end

local displayClass_button = widget.newButton{
    defaultFile = "selectButton.png",
    overFile = "selectButton.png",
```

```

        label = " Return to Menu",
        size = 24,
        emboss=true,
        onRelease = displayClass_function,
        id = "displayClass"
    }
displayClass_button.x = display.contentWidth/2+100
displayClass_button.y = display.contentHeight-200

-- add the button 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)
end --create Function

function scene:show(event)
    localGroup.alpha = 1
end

function scene:hide(event)
    localGroup.alpha=0

end

-- "create" is called whenever the scene is FIRST called
scene:addEventListener( "create", scene )

-- "show" event is dispatched whenever scene transition has finished
scene:addEventListener( "show", scene )

-- "hide" event is dispatched before next scene's transition begins
scene:addEventListener( "hide", scene )

return scene

```

And there we have a basic app that can write and read from a local database.

## **Summary**

Did I happen to mention that working with databases with your program is considered one of the dividing lines between beginning and experienced programmers? Congratulations, you have leveled up! 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.

## **Assignments**

- 1) Modify Project 14.1 to include the student's gender and age.
- 2) Create a database app to store the current date and temperature. The retrieve page should list the dates and temperature.
- 3) Create a high score app that accepts the name of the game, name of player, high score, and the date the high score was achieved.
- 4) Augment Project 14.1 to include the ability to update student information (advanced project).

# Chapter 15 Network Communications

## Learning Objectives

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:

- Determine network communication
- Create a network connection
- Connect to a webserver
- Download from a webserver
- Connect to Facebook
- Introduce functions for in-app ads

## 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 have a framework that anticipates these potential networking issues. While it isn't obvious when you are programming due to the efficiency of smart phones and tablets, Corona is handling two types of networking: cellular network and wireless (WiFi). The basic network connectivity and reachability can be viewed through the `networkStatus` event. The `networkStatus` event can give you information through various properties. To determine the `networkStatus`, you can use the `network.setStatusListener` API:

```
network.setStatusListener(url, listener-function)
```

Currently the listener only supports named URL's (i.e., you can't use an IP address). To stop the listener, just call the the API again, passing `nil` in place of the listener function.

The `networkStatus` events can return the current status of:

- `event.isReachableViaWiFi` – returns true if the URL can be reached via WiFi.
- `event.isReachableViaCellular` – returns true if the URL can be reached via cell network.
- `event.address` – returns the URL address that is being monitored
- `event.isInteractionRequired` – returns true if the user needs to interact with the app to complete or stay connected (i.e., they need to enter a password).
- `event.isConnectionOnDemand` – returns true if the connection will be made automatically.
- `event.isReachable` – returns true if the host can be reached.
- `event.isConnectionRequired` – returns true if the connection is active.

Remember, not all tablets have cellular communication. If your app requires a connection to the Internet to work, it is a good idea to check and see if WiFi can reach the desired URL.

## Asynchronous Network Requests

An http request is a standard network call, just like what would happen if you type a web address in-to a web-browser. Corona SDK contains a full feature of network communication features. With the network API you can make asynchronous HTTP or HTTPS (a secure web connection, usually meaning you have to login to the remote website) requests to a URL.

### HTTP

We will begin with the basics of network communication; establishing a connection using hyper-text transfer protocol (http). As most people are used 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, compressed files, sound files, 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.

On Android, you must add the following permission to the "build.settings" file.

```
settings =
{
    android =
    {
        usesPermissions =
        {
            "android.permission.INTERNET",
        },
    },
}
```

## Project 15.0: 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, then display it to the screen. For this project I have placed an image on my webserver 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 button widget. 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 widget = require("widget")

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
```

As you can see, I used network.download for this demonstration. First turn on the network activity indicator. This will turn on the spinner on status bar showing network activity. Now we can set the URL parameter 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/MobileAppDevelopmentCover.png", "GET", networkListener, "MobileAppDevelopmentCover.png",
system.DocumentsDirectory)
native.setActivityIndicator( false )

end

-- Load Button
loadButton = widget.newButton{
    defaultFile = "buttonBlue.png",
    overFile = "buttonBlueOver.png",
    onRelease = loadButtonPress,
    label = "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 15.0 as a LuaSocket project to show the differences.

### Project 15.1: Picture Download – Via Socket Library

Config.lua and build.settings remain the same. All of the changes occur in main.lua.

#### main.lua

We will use three requires in this version: 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 widget = require("widget")

-- 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 downloading now")

    testImage = display.newImage( "MobileAppDevelopmentCover.png",
system.DocumentsDirectory, 10,30);

    testImage.x = display.contentWidth/2

    native.setActivityIndicator( false )

end

-- Load Button
loadButton = widget.newButton{
    defaultFile = "buttonBlue.png",
    overFile = "buttonBlueOver.png",
    onPress = loadButtonPress,
```

```

        label = "Load Picture",
        emboss = true
    }
loadButton.x = display.contentWidth/2
loadButton.y = display.contentHeight - 50

```

## Web Popup

Web Popups allow you to load a webpage (whether local or remote) from within your app. When called, a web popup is 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 or the Mac OS X (xCode) 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 )
```

## Webviews

WebViews load a remote web page into a webView container. They are different from web popups in that they can be moved, rotated and have physics bodies assigned to them. You can also load html pages into the webView locally. The webView must be removed by calling the removeSelf() method.

The syntax for a webview is

```
native.newWebView( CenterX, CenterY, width, height )
```

The URL is assigned with

```
ObjectName:request(URL [,options])
```

## 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 my CCNA teaching certificate to good use). 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.

## Connecting to Proprietary 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 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 statuses. 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 Facebook’s API arguments.

**Returns:**

  - event.name- name of the event (“fbconnect”)
  - event.type – type of event (“request”)
  - event.response – the JSON 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"} )

```

## Advertising Networks

At the time of this writing, Corona supports three ad vendors; inMobi, inneractive, and iAds. Each service has its own requirements. Each ad vendor has their own requirements. Please check the API for specific requirements before implementing ads in your app.

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:** depends on ad service provider.

### **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.

## **Summary**

In this chapter, we have examined how to setup basic communications with a webserver or proprietary services. Hopefully you will find these resources a great starting point as you create your network-enabled app.

## **Assignments**

- 1) Modify Project 15.0 to download an image from your own webserver.
- 2) Modify Project 15.1 to download an image or file from your webserver.
- 3) Create an app to update your Facebook status.
- 4) Add an ad network to an app that is ready to be published.
- 5) Add a webpopup to an existing app.
- 6) Create an app to handle simple web navigation using the webview API.

# Chapter 16: Head in the Cloud

## Learning Objectives

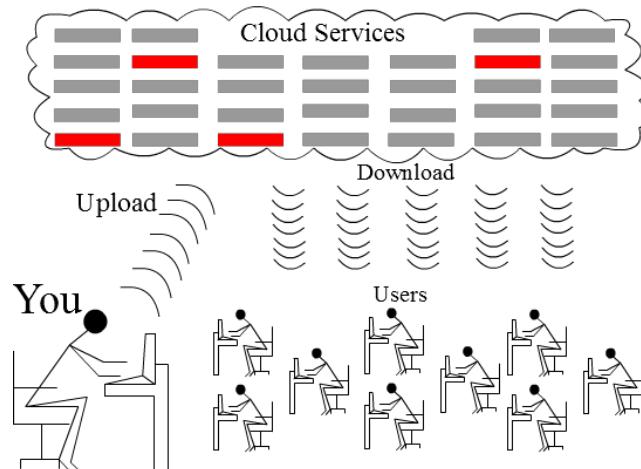
Everywhere we turn today it seems someone in technology is talking about “Cloud” this and “Cloud” that. Utilizing the Cloud has become a very important aspect of mobile and traditional computing. In this chapter we will:

- Explain what the Cloud is and how it is used
- Discuss the levels of implementation of Cloud computing
- Examine how Cloud computing could be used in mobile applications or games
- Discuss multi-user/player methods within Corona SDK

## Cloud Computing

There is a lot of hype surrounding cloud computing. The term “Cloud Computing” creates a lot of confusion as to exactly what Cloud Computing is and how it impacts your apps and games. Often times when someone hears the term, they think of things just magically happening on the Internet and they receive the results. Let’s begin by defining Cloud Computing so that we don’t have any misconceptions. Cloud Computing is using the Internet to access other’s computing resources to complete your task. Really. That’s all it is. Think of it like voice mail on your phone. When you go to check your voice mail, it is located at a remote location and you contact that location to get your messages.

Cloud Computing can vary greatly in what service is being provided by the remote computers. It might be as simple as using Gmail, Hotmail, or Yahoo for your email service or as complex as running a full virtual server remotely through Amazon, Google, or Rackspace.

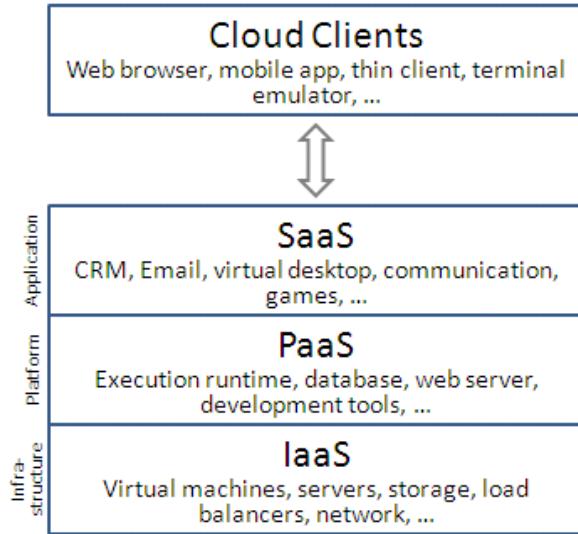


Amazon was one of the first places to begin offering Cloud services with their AWS. It came about through Amazon building an incredible distributed computing server base; large enough to handle Christmas orders on their website without hiccups or delays. Amazon realized that while they needed that much computing power in November and December, the remainder of the year it just sat idle. So they started leasing some of that computer power to others at very reasonable costs. Unlike traditional webhosting where you pay a flat fee for your website, in Cloud Computing, you only pay for what you use in memory, processing, and storage.

One of the things that differentiates Cloud Computing from traditional webhosting is that Cloud Computing is Scalable. Scalable means that if you suddenly have a lot of network traffic, more computer processing power, network bandwidth, memory and whatever else you need, is made available to your website or app as it is needed. This is because your website or app isn't on one particular computer; it is using shared resources: potentially thousands of computers all interconnected and sharing resources.

Cloud Computing is divided into three general categories:

- SaaS – Software as a Service – provides access to software through your web browser or email software to such resources as email, games, virtual desktops, etc. Examples include Google Apps, Microsoft Office 365, and Salesforce
- PaaS – Platform as a Service – provides a virtual server configured for webhosting, database access, development tools, etc. Examples include AWS Elastic Beanstalk, Google App Engine, and Windows Azure Cloud Services.
- IaaS – Infrastructure as a Service – is the most basic model, provides access to a cloud of computer resources that you can use to create your own virtual server with your choice of operating system and resources. Examples include Amazon EC2, Google Compute Engine, and Rackspace.



You might be thinking, “Great! Now I understand what a Cloud really is... so what?” I’m glad you asked! When making a mobile app or game, often times you will want or need to take advantage of cloud services to help it be successful. While there are MANY Cloud Computing services available, we are going to look at a few of the most common for your app and game development needs.

## Game Services

Both Apple and Google Play game services use the gameNetwork API. While Apple’s game center is part of the basic Corona SDK, Google Play is a plugin and requires either a Pro or Enterprise License to use.

The gameNetwork API varies according to its use and includes:

- `gameNetwork.init( center [, listener] )` – initiates game services. *Center* will be either “gamecenter” for Apple, or “google” for Google Play.
- `gameNetwork.request(command [, params])` – sends to or requests information from the game service. Specific *command* differ depending on game services provider and are included under the Apple & Google sections below.
- `gameNetwork.show(command [, data])` – sends to or requests information from the game service. Specific *command* differ depending on game services provider and are included under the Apple & Google sections below.

Using the gameNetwork API will enable Corona Launchpad, which provides feedback and the use of analytics for your app or game.

## Apple Game Center

The Apple Game Center is only available on iOS devices (no Android devices and cannot be used on the Corona Simulator). To use the Apple Game Center, you will need to configure the cloud services through iTunes Connect Portal for your app prior to making network calls.

```
gameNetwork.init( "gamecenter" [, initCallback] ) -- Initializes iOS
```

Specific resources available through the game center include:

- `gameNetwork.request(command [, params])`
  - commands include:
    - `setHighScore` – used to set a high score for current player. If the score is not higher than the one stored on the server, the server will keep the highest value.
    - `loadScores` – returns a table based upon playerID, category, or other data.
    - `loadLocalPlayer` – returns the current player record from Game Center.
    - `loadPlayers` – returns a list of players with specific player IDs.
    - `loadFriends` – returns the friends of the currently logged in user.
    - `loadAchievements` – returns the users completed achievements.
    - `unlockAchievement` – unlocks the specified achievement.
    - `resetAchievements` – resets all achievements for the current player. There is no undo for this!
    - `loadAchievementDescriptions` – returns descriptions of the associated achievements.
    - `loadFriendRequestMaxNumberOfRecipients` – returns Apple's limit on friend requests.
    - `loadLeaderboardCategories` – returns a list of leaderboard categories for the app.
    - `loadPlayerPhoto` – returns the players image from the Apple server.
    - `loadAchievementImage` – returns the achievement image.
    - `loadPlaceholderCompletedAchievementImage` – returns a placeholder image generated by Apple for the requested achievement.
    - `loadIncompleteAchievementImage` – returns the Apple placeholder image for incomplete achievements.
- `gameNetwork.show(command [, params])` – commands include:
  - “leaderboards” – returns leader board.
  - “achievements” – returns achievements.
  - “friendRequest” – returns friend requests.

The `params` table will depend upon the command. Corona has been careful to use the same terminology as Apple Game Center to help avoid confusion. For current details on all parameters available for Apple Game Center, check the Corona Docs gameNetwork API.

### Set High Score:

```

gameNetwork.request( "setHighScore",
{
    localPlayerScore = { category="com.burtonsmmediagroup.myGame", value=25 },
    listener=requestCallback
})

```

### **Load High Scores:**

```

gameNetwork.request( "loadScores",
{
    leaderboard =
    {
        category="com. burtonsmmediagroup.myGame",
        playerScope="Global", -- Global, FriendsOnly
        timeScope="AllTime", -- AllTime, Week, Today
        range={1,5}
    },
    listener=requestCallback
})

```

## **Google Play Game Services**

The Google Play Game Services are only available through the plugin (i.e. only available for those with Pro or Enterprise License) and is only available for Android devices. It does not run on the Corona Simulator. At the time of this writing, this is a relatively new service, so be sure to check the Corona API for improvements.

```
gameNetwork.init( "google" [, initCallback] ) -- Initializes Google Play
```

Specific resources available through the game center include:

- `gameNetwork.request(command [, params])`
  - commands include:
    - `setHighScore` – used to set a high score for current player. If the score is not higher than the one stored on the server, the server will keep the highest value.
    - `loadScores` – returns a table based upon playerID, category, or other data.
    - `loadLocalPlayer` – returns the current player record from Game Center.
    - `loadPlayers` – returns a list of players with specific player IDs.
    - `loadFriends` – returns the friends of the currently logged in user.
    - `loadAchievements` – returns the users completed achievements.
    - `unlockAchievement` – unlocks the specified achievement.
    - `loadAchievementDescriptions` – returns descriptions of the associated achievements.
    - `loadLeaderboardCategories` – returns a list of leaderboard categories for the app.
    - `isConnected` – returns if the user is currently logged into Google Play game services.
    - `login` – attempts to log the user into Google Play game services.
    - `logout` – Logs the user out of Google Play game services.

The *params* table will depend upon the command. Corona has been careful to use the same terminology as Apple Game Center to help avoid confusion. For current details on all parameters available for Google Play game services, check the Corona Docs gameNetwork API.

## Set High Score

```
gameNetwork.request( "setHighScore",
{
    localPlayerScore = { category="Cy_sd893DEewf3", value=25 },
    listener=requestCallback
})
```

## Load High Scores

```
gameNetwork.request( "loadScores",
{
    leaderboard =
    {
        category="Cy_SLDWING4334h",
        playerScope="Global",      -- Global, FriendsOnly
        timeScope="AllTime",       -- AllTime, Week, Today
        range={1,5},
        playerCentered=true,
    },
    listener=requestCallback
})
```

## Pubnub

Pubnub is a nice little cloud tool 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”). They have reasonable plans starting at \$15 per month. Pubnub is not connected to Corona Labs (or myself) in anyway. It is a resource I found that I thought could be useful to app developers. For more information on pubnub, go to <http://www.pubnub.com> or <https://github.com/pubnub/pubnub-api/tree/master/lua-corona>.

## Project 16.0 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. Be sure to copy the pubnub.lua and json.lua files to your folder. You can get these files from either the sample project folder or the pubnub website. 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 9 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.

```
-- initialize 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 = "Ch16-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 Connection!")
    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 = "Ch16-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 9. 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 initial 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
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 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.

## Corona Cloud

You might be wondering why I haven't discussed Corona Cloud. Well, because it's no longer available as a service from Corona. It was only available for less than 6 months then was discontinued because Google and Apple really improved what they were offering through their respective Cloud resources. So rather than re-inventing the wheel, they instead changed their focus to build upon what was already available. Corona Cloud ran on Amazon's AWS structure to provide basic multi-user/player resources such as lobbies, leaderboards, and connections services.

## Summary

It cannot be emphasized enough that cloud computing is a rapidly evolving area. You should always check the current documentation for the newest features that are available.

### **Questions:**

- 1) What is Cloud Computing?
- 2) Define a “scalable” computing resource.
- 3) What is a SaaS?
- 4) What is a PaaS?
- 5) What is a IaaS?
- 6) What is the difference between Cloud Computing and hosting your website with a traditional Internet Service Provider?
- 7) What are some advantages of Cloud Computing?
- 8) What are some disadvantages of Cloud Computing?
- 9) List two uses of Saas.
- 10) Explain why someone might select IaaS over Paas.

### **Assignments:**

- 1) Modify the pubnub project and deploy it to two devices so that the send and receive are on different channels.
- 2) Modify the Star Explorer project from chapter 9 to have a high score board using either Google Play or Apple Game Center.
- 3) Create a pubnub project to send secret messages between two devices.
- 4) Challenge: Add Google Play or Apple Game Center services to one of your apps.
- 5) Big Challenge: Create a multiplayer game.

# Chapter 17 Web-based App Development

## Learning Objectives

Web-based applications are slowly growing in use for mobile devices. In this chapter we will :

- discuss advantages and disadvantages of Web-based apps
- examine how to develop apps that take advantage of the Internet
- define analytics or metrics
- discuss when to use analytics in your app or game.
- use Corona to develop a Web-based application that uses analytics

## When to Build a Web-based App

Web-based applications have been predicted to be the future of mobile application development for several years. It seems that a month doesn't go by where I don't receive a newsletter or magazine with an article with this prediction. At the time of this writing (and more than five years preceding the writing of this book), it is still a prediction. While we are beginning to see more apps that take advantage of the Internet, true Web-based apps are still far from being the dominate approach.

One of the primary reasons that Web-based application development continues to be predicted as one of the greatest things since slice bread is the cross-platform approach of Web-based apps. The app is generally designed to run as a web page rather than to be truly located on the mobile device. Since it is a web page, as long as the mobile device supports a browser that can run the app, the operating system doesn't matter. We already see many examples of Web-based apps and games through networks such as Google's Chrome apps.

Sounds great, doesn't it? Who wouldn't want to write once and deploy to the world!?! Sadly, there are many reasons for the slow adoption. Some of the reasons include: HTML5 approval, HTML5 performance, the rural issue, and WiFi vs. cell. Let us examine each of these issues in turn:

- HTML5 standard - (the proposed standard of Web-based apps) is not due for final approval until 2014. Until finalized by W3C (World Wide Web Consortium – the web standards body), many developers will be slow to make the transition.
- HTML5 performance - in the early versions performance was lacking. These issues have largely been addressed, and while this slowed past adoption, it shouldn't be an issue once the standards are finalized.

- The Rural Issue – is the problem of a lack of web connectivity when traveling in rural parts of the world. Web-based apps need bandwidth. If you are in an area with limited data access, or on a cell plan with a low data allowance, as a user you will probably avoid Web-based apps.
- WiFi vs Cell – is the issue of many mobile devices (specifically tablets) are not on a data plan. Thus, if you are not in a WiFi hotspot, you will not have access to the Web-based app.

## Considerations in Web-based App Development

In my opinion, the final two issues will be the factors that keep Web-based apps from becoming the dominate method of app and game development. I was recently reminded of the rural issue when visiting a rural area. While the place where I was staying offered WiFi, a storm on our second evening there knocked out the DSL line. When the service provider was contacted the next morning, we were told that no one would be available until the following Monday.

Thus, due to a small storm, I was without Internet for my laptop and tablet. While I still had my smartphone, it ensured that I would not be using Web-based apps like Google Docs to do any writing that weekend. More importantly, it also meant I could not play any Internet based games or use Steam or Battle.net since I had not previously configured my account for offline access!

I provide this example to help you think about your application or game from an accessibility stand point. If you want those who use your app to always have access to the app, it must reside on the device and not require Internet access on startup. As an example, during my weekend without Internet, though I did not have access to the web on my tablet, I was able to open previously downloaded emails, write a response, and queue it for sending once our service was restored or when I traveled to someplace that did offer WiFi service. If my email was only available through a browser, this would not have been possible. Since the mail app stored messages (and checked for new messages regularly), I was able to continue with a limited level of productivity.

When designing an app that will require information from the Internet, especially apps that will target tablet or rural users, keep in mind that your users might have limited or periodic access. Adding the capability to download and save information locally, save responses until the user again has Internet access, or has an ‘offline’ mode will greatly enhance your standing within the app using community.

## You say Metrics, I say Analytics

While considering Internet access, it seems appropriate to add the consideration of metrics or analytics. Metrics or Analytics is the process of determining usage app or game usage to measure efficiency, performance, or progress within an app or game.

Recently, analytics gathering has come under fire for gathering data without the user being aware. While traditional methods of data gathering do not contain personally identifiable information, many app/game developers were gathering data without letting the user know that data was being collected. This is a big no-no. If you are gathering data, you need to make sure your users are aware of it (usually a statement in the Terms & Agreements is sufficient).

Analytics is handled through the analytics API (don't you love easy to remember API calls?). Currently the only service available through Corona is Flurry. You can signup for a Flurry account at <http://www.flurry.com>. The API has two commands:

- `analytics.init(apikey)` – Initializes the analytics.
- `analytics.logEvent(eventID [,params])`

After you sign up for your Flurry account, you will be able to create your application on Flurry's website. You can choose iPhone, iPad, or Android. This will create an application key which you must enter in the `analytics.init` command.

It should be noted that Flurry now offers a restricted analytics option that is appropriate for use in apps or games that target children. The data gathered is restricted in the type of information gathered and does not include anything that could potentially identify the person using the app but still provides basic usage information.

Note that user permissions for Android must be enable in your build.settings file:

`build.settings`

```
settings =
{
    android =
    {
        usesPermissions =
        {
            -- This permission is required in order for analytics
            to be sent to Flurry's servers.
            "android.permission.INTERNET"
        },
    },
}
```

```
}
```

A simple example of using Flurry would be:

main.lua

```
local analytics = require "analytics"

-- initialize Flurry analytics with your app key
analytics.init( "YOUR_API_KEY" )

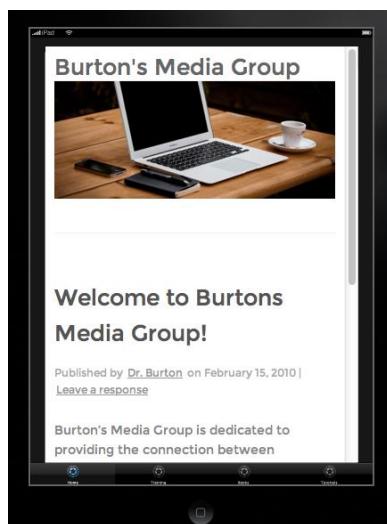
-- log your events
analytics.logEvent( "Event ID" )
```

We will use Flurry in our next app so that you can see it in action.

## Project 17: Web-Hybrid app

A very common type of app that is requested is a migration of a website to an application. As I mentioned previously, the greatest problem with this type of app is the loss of Internet connectivity. Fortunately a web-hybrid app takes advantage of the ability to store the last successful download locally should an Internet connection not be available. We are also going to include basic analytics in this app to show page navigation during the apps use.

This app will pull four pages from my website which uses WordPress. To improve the appearance of the website on mobile devices I am using a plugin called **Theme Test Drive** to load a theme called **Plain-WP**.



config.lua

```

application =
{
    content =
    {
        width = 768,
        height = 1024,
        scale = "letterbox",
        antialias = false,
        xAlign = "center",
        yAlign = "center"
    }
}

```

Nothing particularly new about the config.lua file beyond using a larger width and height setting (the default size for a first generation iPad).

## build.settings

```

settings =
{
    android =
    {
        versionCode = "1"
    },
    androidPermissions =
    {
        "android.permission.INTERNET"
    },
    orientation =
    {
        default ="portrait",
        supported =
        {
            "landscapeLeft", "landscapeRight", "portrait", "portraitUpsideDown"
        },
    },
    build =
    {
        custom = "1",
    }
}

```

```
}
```

You must setup the permission for Android if you plan to deploy to that operating system.

To get started on our main program we will need to setup our variables and load our widget & analytics. I am declaring most of the variables that will be used in the program at the front to make calling them easier later in the program. Several of the variables such as the webviews are used then removed then used again for orientation change. By declaring them at the beginning, we can avoid later problems with variables that are local to specific functions.

### main.lua

```
local analytics = require("analytics")
local widget = require("widget")
local webView, tabBar, tabButtons
local splash, loading, spinner
local bkgd, orient, current_page
local webView_home, webView_books, webView_training,
webView_tutorial
```

We will use a tab bar to move between the pages in the app. By setting up a routine for each page, we can hide the pages not in use by setting their alpha to zero. I am also logging an analytics event as to which page is loaded. We will also set a variable called current\_page which is used should an orientation change event happen so that we will know which page to reload. All four pages are loaded from where they have been saved in the system.DocumentsDirectory.

```
local function home()
    webView_home:request("home.html",
system.DocumentsDirectory)
    print("home")
    analytics.logEvent("Home")
    current_page = "home"
    webView_home.alpha=1
    webView_books.alpha=0
    webView_training.alpha=0
    webView_tutorial.alpha=0
    return true
end

local function training()
    webView_training:request("classes.html",
system.DocumentsDirectory)
    print("training")
    analytics.logEvent("Training")
    current_page = "training"
```

```

        webview_home.alpha = 0
        webview_books.alpha=0
        webview_training.alpha=1
        webview_tutorial.alpha=0
        return true
    end

    local function books(event)
        webview_books:request("books.html",
        system.DocumentsDirectory)
        print("books")
        analytics.logEvent("Books")
        current_page="books"
        webview_home.alpha = 0
        webview_books.alpha=1
        webview_training.alpha=0
        webview_tutorial.alpha=0
        return true
    end

    local function tutorials()
        webview_tutorial:request("blog.html",
        system.DocumentsDirectory)
        print ("Tutorials")
        analytics.logEvent("Tutorials")
        current_page = "tutorials"
        webview_home.alpha = 0
        webview_books.alpha=0
        webview_training.alpha=0
        webview_tutorial.alpha=1
        return true
    end

```

The loadPages function is only called the first time the pages are loaded. It creates the four webviews and sets the alpha to all but the home webview to hidden. After setting up the webviews, we setup the tab bar. Notice that the webview leaves space for the tab bar to be at the bottom of the screen. Remember that webviews are native objects, so they will automatically be on top of everything else on the screen. Thus, we have to make it a smaller area of the screen to leave room for the tab bar. The last command of this function is to call the home function above, which loads the app homepage.

```

local function loadPages()
    webview_home = native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)

```

```

    webview_books =
native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webview_training = native.newWebView(display.contentCenterX
+20, display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webview_tutorial =
native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webview_books.alpha=0
    webview_training.alpha=0
    webview_tutorial.alpha=0
current_page = "home"

splash.alpha =0
splash:removeSelf()
loading:removeSelf()
spinner:removeSelf()

--create tab bar for navigation
tabButtons = {
{
    width =32,
    height = 32,
    defaultFile = "tabIcon.png",
    overFile = "tabIcon-down.png",
    label = "Home",
    onPress = home,
    selected = false},
{
    width =32,
    height = 32,
    defaultFile = "tabIcon.png",
    overFile = "tabIcon-down.png",
    label = "Training",
    onPress = training,
    selected = false},
{
    width =32,
    height = 32,
    defaultFile = "tabIcon.png",
    overFile = "tabIcon-down.png",
    label = "Books",
    onPress = books,
    selected = false},

```

```

    {
        width =32,
        height = 32,
        defaultFile = "tabIcon.png",
        overFile = "tabIcon-down.png",
        label = "Tutorials",
        onPress = tutorials,
        selected = false},
    }

tabBar = widget.newTabBar {
    top = display.contentHeight -50,
    width = display.contentWidth,
    buttons = tabButtons
}
tabBar:selected(1)

home() -- load the home page
end

```

The networkListerner function is called by the initial call to load each of the four webpages. When the home pages is loaded and saved to the system.DocumentsDirectory, the function will call the loadPages function above and eventually the home function.

```

local function networkListener( event )
    if ( event.isError ) then
        print( "Network error, load locally")
        local warning = "You are not currently connected
to the Internet. This is a locally stored page."
    else
        print ("Connected to Server")
    end
    if(event.response.filename == "home.html") then
        loadPages()
    end
end

```

SplashScreen will load our intial screen that will show while the webpages are downloading from the Internet. I have created two splash screens for this app, one for portrait view and one for landscape. We also start the spinner widget in this function so that the user knows the app is attempting to download new material.

```

local function splashScreen()
    if(system.orientation == "portrait" or
system.orientation == "portraitUpsideDown") then

```

```

        bkgd=display.newImage("Default-Portrait.png",0,20)
        splash = display.newImage("splashp.png",0,0)
    else
        bkgd=display.newImage("Default-Landscape",0,20)
        splash = display.newImage("splashl.png",0,0)
    end
    loading = display.newImage("loading_banner.png",0,0)
    loading.x = display.contentWidth-(loading.width/2+35)

    loading.y = 100
    spinner = widget.newSpinner {
        left=display.contentWidth-40,
        top = 70,
    }
    spinner:start()
end

```

If we are going to have multiple orientations, it makes sense to have a routine to handle orientation change! Since we are working with native objects, I found it easier to remove the four webviews and re-create them to the new orientation size. I also reloaded the tab bar so that it would be the correct size and location for the new orientation.

The final part of this function looks at which page is currently loaded and reloads the page into the new webview.

```

local function onOrientationChange()
    tabBar:removeSelf()
    webView_home:removeSelf()
    webView_books:removeSelf()
    webView_training:removeSelf()
    webView_tutorial:removeSelf()
    webView_home = native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webView_books =
native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webView_training =
native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webView_tutorial =
native.newWebView(display.contentCenterX+20,
display.contentCenterY+20,display.contentWidth-
66,display.contentHeight-90)
    webView_home.alpha=0

```

```

webview_books.alpha=0
webview_training.alpha=0
webview_tutorial.alpha=0
tabBar = widget.newTabBar {
    top = display.contentHeight -50,
    width = display.contentWidth,
    buttons = tabButtons
}
if(current_page=="home") then
    tabBar:selected(1)
    home()
elseif(current_page=="training") then
    tabBar:selected(2)
    training()
elseif(current_page=="books") then
    tabBar:selected(3)
    books()
else
    tabBar:selected(4)
    tutorials()
end

end

```

We are done with our functions. Now it is time to get the app going. We will start by initializing our Flurry analytics and calling the splashScreen function so that something shows on the screen while we attempt to load the webpages. Each webpage will need its own set of parameters.

You might notice in the network.request that I have added a little bit to the URL call. This is an easy way to utilize the WordPress plugin that I mentioned earlier to load a special theme just for the app.

```

analytics.init("Place your Flurry Key Here") -- no, I'm not
sharing my key with you :-
splashScreen()

local params={}
params.response = {
    filename = "home.html",
    baseDirectory = system.DocumentsDirectory
}
network.request("http://www.burtonsmediagroup.com/?theme=plain-
wp", "GET", networkListener, params)

local paramsBooks = {}

```

```

paramsBooks.response = {
    filename = "books.html",
    baseDirectory = system.DocumentsDirectory
}
network.request("http://www.burtonsmidiagroup.com/books/?theme=plain-wp", "GET", networkListener, paramsBooks)

local paramsTraining = {}
paramsTraining.response = {
    filename = "classes.html",
    baseDirectory = system.DocumentsDirectory
}
network.request("http://www.burtonsmidiagroup.com/classes/?theme=plain-wp", "GET", networkListener, paramsTraining)

local paramsBlog = {}
paramsBlog.response = {
    filename = "blog.html",
    baseDirectory = system.DocumentsDirectory
}
network.request("http://www.burtonsmidiagroup.com/blog/?theme=plain-wp", "GET", networkListener, paramsBlog)

Runtime:addEventListener("orientation", onOrientationChange)

```

As you can see, it is fairly easy to create a hybrid app that allows your users to have access to the website even when they do not have Internet access, allowing your app to have the best of both worlds!

Note: For analytics to work, the app must be deployed to a device.

## Summary

Hybrid apps will be one of the fastest growing and rapidly changing areas of mobile development. You will want to watch the development of HTML5 closely and consider how that impacts your app. By creating hybrid apps, you can leverage the constantly changing nature of the web with the reliability of an app.

## Questions

- 1) When is a web-only app most appropriate?

- 2) When would you want to create an app that is not web-enabled?
- 3) At what point is a hybrid app most appropriate?
- 4) Define HTML5.
- 5) Define analytics.
- 6) When is analytic use appropriate?
- 7) Research what types of information is illegal to gather in analytics for apps targeting children.
- 8) What types of information would be useful to you as an app/game developer to gather from analytics?
- 9) What are the ethical/moral implications of gathering analytics about your app/game users?
- 10) List three advantages for web-only, apps with no Internet connection, and hybrid apps.

## Assignments

- 1) Modify Project 17 for your website or a favorite website.
- 2) Add additional analytic events that would be useful to Project 17.
- 3) Add analytics to a previous app that you have created.
- 4) Modify Project 17 to include additional webpages.

# Chapter 18: Advanced Graphics (and a game)

## Learning Objectives

In this final chapter of using Corona SDK we will examine Graphics 2.0 and create a simple side scrolling game (SSSG for short). I thought long and hard about what to do for this final chapter and then asked the twitter-verse. The overwhelming response was for a side scrolling game. So here it is! It seemed appropriate to also discuss some of the capabilities of Graphics 2.0 at the same time:

- Filters, Generators, and Composites
- Liquid Fun
- Learn about parallax scrolling
- Adding a distance meter
- Use physics to simulate jumping

## Graphics 2.0

Graphics 2.0 was a major update to the graphics engine for Corona SDK in late 2013. It moved the base graphics engine to OpenGL ES 2.0. This added all types of new capabilities, a major improvement to performance, and a shader-based pipeline for doing some interesting things with graphics.

### Paint

Before we dive too deep into some of the special effects available, we need to take a moment to discuss paint. There are several different uses and applications of paint in Corona. The most fundamental is, just as the name implies, like paint. You can select a color using either the traditional {red, green, blue [, alpha]}, or limit it to gray {gray [, alpha]} with the range of each color between 0 and 1.

Paint is not limited to solid colors. You can also use:

- BitmapPaint - which allows you to use a second display object (such as an image) as your fill or image stroke.
- CompositePaint – which allows the selection of multiple textures/images to be used as fills and strokes.
- GradientPaint – used to provide a linear gradient fill or stroke.
- ImageSheetPaint – used to draw your fill or stroke from a specific frame on an image sheet.

## Fills - Filters, Composite, and Generators

One of the most exciting tools that is now available to developers is the use of filters, composite, generators to change or augment your graphics with a minimum of effort. Each of these is slightly different, so let's look at how to apply each of them separately.

### Fills

Fills are just like they sound, they 'fill' a vector shape with an image, composite, or effect. To fill an object, create the vector image (display.newCircle, display.newRect, etc), then use the fill property on the object:

```
1 local myObject = display.newCircle(0, 0, 256)
2 myObject.x = display.contentCenterX
3 myObject.y = display.contentCenterY
4 myObject.fill = {type = "image", filename = "Myphoto.png"}
```

The fill parameters for type are image, composite, and gradient.

### Filters

There are so many filter effects available! All filter effects are applied to a single image or texture by using the effect property of object.fill or object.stroke (i.e. myObject.fill.effect = "filter.effect" or myObject.stroke.effect = "filter.effect").

At the time of writing, there are 45 effects available:

- filter.bloom – increases the light saturation of bright areas in an image. Parameters: levels.white (0 to 1; default =0.843), levels.black (0 to 1; default =0.565), levels.gamma(0 to 1; default =1), add.alpha (0 to 1; default =1), blur.horizontal.blurSize (2 to 512; default =8), blur.horizontal.sigma (2 to 512; default =128), blur.vertical.blurSize (2 to 512; default =8), and blur.vertical.sigma (2 to 512; default =128).
- filter.blur – as the name implies, the image will be blurred.
- filter.blurGaussian – A blur function based upon Gaussian function, resulting in a stronger blur effect in most situations. Parameters: horizontal.blurSize (2 to 512; default =8), horizontal.sigma(2 to 512; default =128), vertical.blurSize (2 to 512; default =8), vertical.sigma (2 to 512; default =128).
- filter.blurHorizontal – A blur effect creating a strong side to side effect. Parameters: blurSize (2 to 512; default =8) and sigma (2 to 512; default =8).
- filter.blurVertical – A blur effect that creates a up and down streaking effect. Parameters: blurSize (2 to 512; default =8) and sigma (2 to 512; default =8).

- [filter.brightness](#) – Brightens the image. Parameter: intensity (0 to 1; default =0).
- [filter.bulge](#) – Creates a lens bulging effect of either concave or convex direction. Property: intensity (0 to unlimited; default =1). Values of less than 1 result in concave (inward bluge), while greater than 1 will result in a convex (outward bulge).
- [filter.chromaKey](#) – Sets a portion (or all) of the image to clear (or an alpha of 0) based upon the selected color. Parameters: sensitivity (0 to 1, .4 default), smoothing (0 to 1, 0.1 default), and color (RGBA table). See Project 18.1 below for an example of using the chromaKey filter.
- [filter.colorChannelOffset](#) – Moves or separates the colors of the image the specified number of pixels. Parameters: xTexels and yTextels (offset in pixels).
- [filter.colorMatrix](#) – Multiplies a source color and adds a bias or offset to the image. Parameters: coefficients (4x4 table of RGBA) and bias (RGBA between -1 and 1).
- [filter.colorPolynomial](#) – Applies a cubic polynomial to the image. Parameter: coefficients, (4x4 table of RGBA).
- [filter.contrast](#) – Increases or decreases the image contrast. Parameter: contrast, (0 to 4; default =1).
- [filter.crosshatch](#) – Creates a crosshatch design based upon the original image. Property: grain (0 to unlimited; default =0).
- [filter.crystallize](#) – Creates a blurred/crystalized effect based upon the number of tiles or crystals specified. Parameter: numTiles (2 to unlimited; default =16).
- [filter.desaturate](#) – Shifts the colors toward the gray scale. Parameter: intensity (0 to 1; default =0.5).
- [filter.dissolve](#) – Breaks up the image. Parameter: threshold (0 to 1; default =1).
- [filter.duotone](#) – Adds tone of a grayscale image. Parameters: darkColor (RGBA table), lightColor (RGBA table).
- [filter.emboss](#) – Creates a greyscale emboss of the image. Parameter: intensity (0 to 4; default =1).
- [filter.exposure](#) – Changes exposure or light saturation of the image. Parameter: exposure (-10 to 10; default =0).
- [filter.frostedGlass](#) – Applies a blur effect simulating frosted glass. Parameter: scale (1 to unlimited; default =64).
- [filter.grayscale](#) – Changes a color image into a grayscale image.
- [filter.hue](#) – Adjusts the hue of the image. Parameter: angle (0 to 360; default =0).
- [filter.invert](#) – Inverts the colors of the image.
- [filter.iris](#) – Changes the center of the image to clear or transparent. Parameters: center (0,0 to 1, 1; default =0.5, 0.5), aperture (0 to 1; default =0), aspectRatio (0 to unlimited; default =1), smoothness (0 to 1; default =0).

- `filter.levels` – Adjusts the black, white, or gamma levels of the image. Parameters: white (0 to 1; default =0.843), black (0 to 1; default =0.565), and gamma (0 to 1; default =1).
- `filter.linearWipe` – Adjusts the image so that a portion of it fades to transparency. Parameters: direction ({-1,-1} to {1, 1}; default ={1, 0}), smoothness (0 to 1; default =0), progress (0 to 1; default =0).
- `filter.median` – Shifts the colors toward the median color in the image, causing a blurring or fading.
- `filter.monotone` – Shifts the image colors toward a specific color. Parameters: r (0 to 1; default =0), g (0 to 1; default =0), b (0 to 1; default =0), a (0 to 1; default =1).
- `filter.opTile` – Applies a tiled effect to the image. Parameters: numPixels (0 to unlimited; default =8), angle (0 to 360; default =0), and scale (0 to unlimited; default =2.8).
- `filter.pixelate` – Creates a pixilation effect on the image. Parameter: numPixels (0 to unlimited; default =4).
- `filter.polkaDots` – Changes the image into appropriately colored dots. Parameters: numPixels(4 to unlimited; default =4), dotRadius (0 to 1; default =1), aspectRatio (0 to unlimited; default =1).
- `filter.posterize` – Shifts the colors of the image to appear more poster like. Parameter: colorsPerChannel ( 2 to unlimited; default =4).
- `filter.radialWipe` – Turns the image to clear in a radial sweep. Parameters: center (-1, -1) to {1,1}; default ={0.5, 0.5}), smoothness (0 to 1; default =0), axisOrientation (0 to 1; default =0), progress (0 to 1; default =0).
- `filter.saturate` – Increases the color saturation of the image. Parameter: intensity (0 to 8; default =1).
- `filter.scatter` – As the name implies, the image becomes scattered or fragmented, the higher the intensity, the more scattered the image becomes. Parameter: intensity (0 to unlimited; default =0.5).
- `filter.sepia` – Transforms the image to appear like an old photograph. Parameter: intensity (0 to 1; default =1).
- `filter.sharpenLuminance` – Transforms the image by sharpening the lighter areas. Parameter: sharpness (0 to 1; default=0).
- `filter.sobel` – Modifies the image to emphasize the edges or transitions.
- `filter.straighten` – As the name implies, this is used to straighten the image. Parameters: width (1 to unlimited; default=1), height (1 to unlimited; default=1), and angle (0 to 360, default=0).
- `filter.swirl` – Creates a swirl effect on the image, making it appear that the center of the image has been twisted or is going down a drain. Parameter: intensity (0 to unlimited; default 0).

- filter.vignette – Darkens the outer edge of the image. Parameter: radius (0 to 1; default=0.1).
- filter.vignetteMask – Instead of darkening the edge like vignette, vignetteMask changes the outer edge to clear (alpha =0). Parameters: innerRadius (0 to 1; default= 0.25) and outerRadius (0 to 1; default=0.8).
- filter.wobble – Creates the appearance of a camera movement or wobble. Parameter: amplitude (none to unlimited; default=10). Note: It is possible to animate the wobble effect using the `display.setDrawMode( "forceRender" )` draw mode. This is very processor-intensive and should be turned off when you are finished with the effect.
- filter.woodCut – Changes the image to appear as if it were a wood cut, showing just the edges of objects. Parameter: intensity (0 to 1; default=0.5).
- filter.zoomBlur – Creates a blur effect simulating zooming in on the image. Parameters: u (horizontal origin – 0 to 1; default=0.5), v (vertical origin – 0 to 1; default=0.5), and intensity (0 to 1; default=0.5).

Whew, that is a lot of possible effects! Here is an example of how to implement one of the effects:

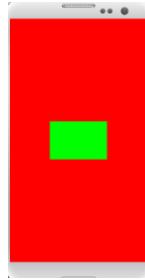
### Project 18.1 Filtering with chromaKey

In this example, we will use the chromaKey filter to set an area to transparent.

```

1 -- create and center object (can be any type of display object)
2 local bkgd = display.newRect( display.contentCenterX, display.contentCenterY,
display.contentWidth, display.contentHeight )
3 bkgd:setFillColor( 1,0,0 )
4
5 local myObject = display.newRect( 20, 20, 300, 200 )
6 myObject:setFillColor(0,1,0,1)
7 myObject.x = display.contentCenterX
8 myObject.y = display.contentCenterY
9
10 -- set color to be made clear with ChromaKey
11 myObject.fill.effect = "filter.chromaKey"
12 myObject.fill.effect.color = {0, 0, 0}
```

In this first version of the program, you should see a green square on a red background.



When we change the value of the last line to:

```
12 myObject.fill.effect.color = {0, 1, 0}
```

The green square becomes transparent.



With more complex images, this has the effect of rendering a portion of the image transparent.

For a visual representation of the impact of different effects, please visit  
<http://docs.coronalabs.com/guide/graphics/effects.html>.

## Composite Effects

Composite Effects are similar to Effects, except that instead of using one image or texture as your fill, you are taking two textures or images, combining them, then the combined paint or fill is applied to an object. As with Effects, we have quite a number of possible Composite Effects available. The process to make a composite paint or fill generally follows this pattern:

```
1 -- Create the object
2 local object = display.newRect( 100, 100, 160, 160 )
3
4 -- Set up the composite paint (distinct images)
5 local compositePaint =
6   type="composite",
7   paint1={ type="image", filename="image01.png" }, -- bottom image
```

```

8 paint2={ type="image", filename="texture.png" }    -- top image
9 }
10
11 -- Apply the composite paint as the object's fill
12 object.fill = compositePaint
13
14 -- Set a composite blend as the fill effect
15 object.fill.effect = "composite.add"

```

The first image loaded (paint1) is considered the bottom image. The second image (paint2) is the top image. Think of it like a stack of photos that you are setting on a table, with the second image set on top of the first, but it can be seen through partially, impacting the appearance of the first (bottom) image.

Available composites include:

- [composite.add](#) – Adds or overlays the second image to the first. Parameter: alpha (0 to 1; default 1).
- [composite.average](#) – Averages the two images. Parameter: alpha (0 to 1; default 1).
- [composite.colorBurn](#) – Darker areas of the second image ‘burn’ or darken the first image. Parameter: alpha (0 to 1; default 1).
- [composite.colorDodge](#) – The light areas of the second image lighten the first or bottom image. Parameter: alpha (0 to 1; default 1).
- [composite.darken](#) – As the name implies, darken shows whichever pixel is the darkest between the two images, resulting in a darker paint or fill. Parameter: alpha (0 to 1; default 1).
- [composite.difference](#) – Subtracts the top image from the bottom or the bottom from the top; whichever results in a positive value. Compositing with a dark top image will result in little to no change; with a light image, the bottom image will be inverted. Parameter: alpha (0 to 1; default 1).
- [composite.exclusion](#) – Like difference, exclusion subtracts the top image from the bottom or the bottom from the top; whichever results in a positive value. Compositing with a dark top image will result in little to no change; with a light image, the bottom image will be inverted but with a lower contrast than difference. Parameter: alpha (0 to 1; default 1).
- [composite.glow](#) – Similar to colorDodge, but as strong of an impact. The light areas of the top image lighten the bottom image. Parameter: alpha (0 to 1; default 1).
- [composite.hardLight](#) – Combines the composite effects of multiply and screen. Dark areas of the bottom image result in a darker top image and light areas of the bottom image result in a brighter top image. Parameter: alpha (0 to 1; default 1).

- `composite.hardmix` – Forces the color channel to 1 for grey areas that are 0.5 (128) or greater and 0 for grey areas that are less than 0.5 (127). Parameter: alpha (0 to 1; default 1).
- `composite.lighten` – Shows whichever pixel is brightest between the two images, resulting a lighter paint or fill. Parameter: alpha (0 to 1; default 1).
- `composite.linearLight` – If the areas of the top image that are light, lighten the bottom image; areas of the top image that are dark, darken the bottom image. Parameter: alpha (0 to 1; default 1).
- `composite.multiply` – Results in a darker bottom image. The darker the top image is, the greater the impact on the bottom image. Parameter: alpha (0 to 1; default 1).
- `composite.negation` – Results in the bottom image becoming transparent wherever the top image is light. The lighter the area of the top image, the closer to an alpha of 0 that portion of the image becomes. Parameter: alpha (0 to 1; default 1).
- `composite.normalMapWith1DirLight` – Applies a rendered directional light to the textures. Parameters: *dirLightDirection* - specifies when and where the light comes from with a value of 0,0,0 being the origin point of left, top corner, on the same layer as the texture. A z value greater than 0 moves the light in toward the user's perspective in virtual space ({x, y, z}, {0,0,0} to {1,1,1}; default {1,0,0}), *dirLightColor* – color of the directional light ({r,g,b,a}, {0,0,0,0} to {1,1,1,1}; default {1,1,1,1}), *ambientLightIntensity* (0 to unlimited; default 0).
- `composite.normalMapWith1PointLight` - Applies a rendered point light to the textures. Parameters: *pointLightPos* - specifies when and where the light comes from with a value of 0,0,0 being the origin point of left, top corner, on the same layer as the texture. A z value greater than 0 moves the light in toward the user's perspective in virtual space ({x, y, z}, {0,0,0} to {1,1,1}; default {1,0,0}), *pointLightColor* – color of the directional light ({r,g,b,a}, {0,0,0,0} to {1,1,1,1}; default {1,1,1,1}), *ambientLightIntensity* (0 to unlimited; default 0), *attenuationFactors* ({0,0,0} to {unlimited, unlimited, unlimited}, default {0.4, 3, 20}).
- `composite.overlay` – Combines the composite effects of multiply and screen. Dark areas of the top image result in a darker bottom image and light areas of the top image result in a brighter bottom image. Parameter: alpha (0 to 1; default 1).
- `composite.phoenix` – Dark areas of the texture reverse the colors of the bottom image, white area is transparent. Parameter: alpha (0 to 1; default 1).
- `composite.pinLight` – Dark areas of the top layer are treated as transparent or having a pin stuck through it, while light areas are treated as opaque or solid depending on the alpha setting. Parameter: alpha (0 to 1; default 1).
- `composite.reflect` - The output of reflect is similar to using a colored light on an object. The end product will show the color saturated with the texture color. Parameter: alpha (0 to 1; default 1).

- `composite.screen` – The opposite of the multiply composite effect. Results in a brighter bottom image. Parameter: alpha (0 to 1; default 1).
- `composite.softLight` – Like hardLight, but the effect is not as intense. Combines the composite effects of multiply and screen. Dark areas of the bottom image result in a darker top image and light areas of the bottom image result in a brighter top image. Parameter: alpha (0 to 1; default 1).
- `composite.subtract` – Subtracts the colors of the top image from the bottom image. Any negative values result in black being displayed. Parameter: alpha (0 to 1; default 1).
- `composite.vividLight` – Combines the composite effects of colorDodge and colorBurn with effect of increasing the contrast of the bottom image. Parameter: alpha (0 to 1; default 1).

## Generators

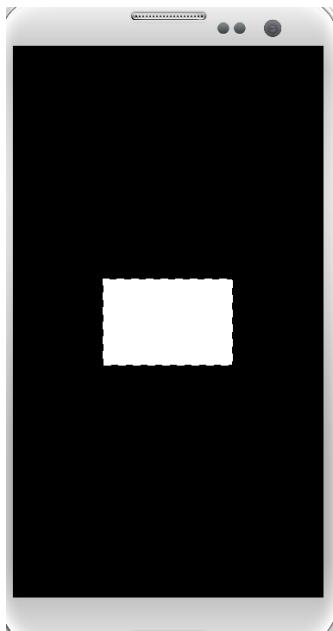
Generators are just what their name implies, they generate a graphic effect automatically. Instead of using a base image to make the change, like with effects, generators are procedurally generated (all programming, no images). These are easy-to-use tools that allow you to augment your app or change images in a simple and straight forward manner. While not as numerous as filters, I think you will see many ways they can be applied to your app.

- `generator.checkerboard` – Creates a checkerboard pattern. xStep and yStep determine the size of each square Parameters: color1 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), color2 ({0,0,0,0} to {1,1,1,1}; default {0,0,1,1}), xStep (1 to unlimited; default 3), and yStep (1 to unlimited; default 3).
- `generator.lenticularHalo` – Applies a halo effect on an object designed to create the illusion of depth. The seed parameter is used to randomly vary the appearance of the halo. Parameters: posX (0 to 1; default 0.5), posY (0 to 1; default 0.5), aspectRatio (0 to unlimited; default 1), and seed (0 to unlimited; default 0).
- `generator.linearGradient` – Applies a linear gradient to the object. Parameters: color1 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), position1 ({0,0} to {1,1}; default {0,0}), color2 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), position2 ({0,0} to {1,1}; default {1,1}).
- `generator.marchingAnts` – Applies a moving dotted line around the selected object. Note: “forceRender” draw mode must be enabled for this effect. This generator has no parameters.

- generator.perlinNoise – Applied to an object to give a more ‘real’ appearance instead of computer generated. Parameters: color1 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), color2 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), scale (0 to unlimited; default 8).
- generator.radialGradient – Applies a radial gradient onto an object. Parameters: color1 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), color2 ({0,0,0,0} to {1,1,1,1}; default {1,0,0,1}), center\_and\_radius – ({center x, center y, inner radius, outer radius}; {0,0,0,0} to {1,1,1,1}; default {0.5, 0.5, 0.125, 0.125}), aspectRatio (0 to unlimited; default 1).
- generator.random – Applies random noise to the object. The generator has no parameters.
- generator.stripes – Applies a stripe pattern to the object. Parameters: periods – table that provides the width in pixels of stripes {w stripe1, w space1, w stripe2, w space2}; default is {1,1,1,1} , angle (0 to 360; default 0), translation – offset of pattern (0 to unlimited; default 0).
- generator.sunbeams – Applies a sunbeam effect to the object. Parameters: posX (0 to 1; default 0.5), posY (0 to 1; default 0.5), aspectRatio (0 to unlimited; default 1), and seed (0 to unlimited; default 0).

## Project 18.2 Marching Ants

One of the most useful generators (at least in my opinion) is the marching ants. With the marching ants generator you can create a stroke outline around an object that moves or ‘marches’ around the object. As mentioned above, to animate the movement of the marching ants, you must set the draw mode to “forceRender” so that the device will redraw the entire scene with each frame refresh. This is a big battery drain and should only be done while the effect is needed.



```
1
2 display.setDrawMode( "forceRender" )
3
4 local myObject = display.newRect(display.contentCenterX, display.contentCenterY, 300, 200)
5 myObject:setFillColor(1,1,1,1)
6
7 -- set up the marching ants
8 myObject.strokeWidth = 2
9 myObject.stroke.effect = "generator.marchingAnts"
```

For a visual representation of the impact of different generators, please visit  
<http://docs.coronalabs.com/guide/graphics/effects.html#generator>.

## Containers

The new graphics engine in Corona gives us another really cool feature: containers. Containers function very similar to groups for managing groups of objects. One of the big differences between groups and containers is that containers are self-masking. In other words, when you use a container, you can set the size of the container. Any object that is a member of that container that leaves the area of the container will be automatically hidden or masked.

Containers allow us to set a stage much easier and more quickly than previous methods. If your app or game is primarily using a specific area of the screen, by setting a container

instead of a group, any object that glitches and leaves the area will automatically be hidden from view.

### Project 18.3 Containers

```
1 local myContainer = display.newContainer( 300, 200 )
2 myContainer.x = display.contentCenterX
3 myContainer.y = display.contentCenterY
4 local gradient = { type="gradient", color1={1,0,0}, color2={0.2,0,0}, direction="down" }
5
6 local bg = display.newRect( 0, 0, 1000, 1000 )
7 bg.fill = gradient
8 myContainer:insert( bg )
9 local myText = display.newText( "Hello World!", 0, 0, native.systemFont, 24 )
10 myContainer:insert( myText )
11
12 local function moveText()
13     transition.to( myText, {time = 2000, x=myText.x+200} )
14 end
15
16 timer.performWithDelay( 2000, moveText )
```

A couple of things to note about containers. First, when you insert an object into a container, giving the coordinates of 0, 0 will place the new object in the **center** of the container. In our example above, myText is placed at 0, 0, which puts it in the center of the myContainer object.

Second, most devices can only handle a maximum of 3 containers active at any point in time. So use containers when you need them, but don't over use them!

### Liquid Fun

While not truly a graphics but a physics tool, this was just too neat (and recently added) to leave out. Liquid fun became available in Corona as of daily build 2322. Released by Google, Liquid fun allows you to simulate fluid in the Box2D engine. There are a whole host of examples of using Liquid Fun in the Corona SDK/Sample Code/Physics folder. More examples are available at <https://google.github.io/liquidfun/>.

For this project we are going to modify one of the sample code examples.

### Project 18.4 Liquid Fun

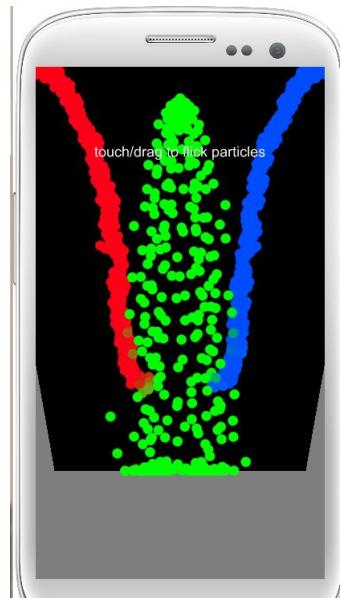
```
1 --Liquid Fun Example
2
3 local physics = require( "physics" )
4 physics.start()
5
```

```

6 display.setStatusBar( display.HiddenStatusBar )
7 display.setDefault( "fillColor", 0.5 )
8
9 -- Add all physics objects
10 local left_side_piece = display.newRect( -40, display.contentHeight-220, 400, 70 )
11 physics.addBody( left_side_piece, "static" )
12 left_side_piece.rotation = 80
13
14 local center_piece = display.newRect( display.contentCenterX, display.contentHeight-16,
15 400, 120 )
16 physics.addBody( center_piece, "static" )
17 local right_side_piece = display.newRect( display.contentWidth+40, display.contentHeight-
18 220, 400, 70 )
19 physics.addBody( right_side_piece, "static" )
20 right_side_piece.rotation = -80

```

We start by initializing the physics engine, hiding the status bar and setting a default fill color. The physics objects in lines 10 through 19 serve as boundaries for the liquid so that it can pool and mix at the bottom. Next we will setup the particle system used to generate the simulated liquid.



```

21 -- Create the particle system
22 local particleSystem = physics.newParticleSystem{
23   filename = "particle.png",
24   colorMixingStrength = 0.1,

```

```

25 radius = 3,
26 imageRadius = 6
27 }
28
29 -- Paramaters for red particle faucet
30 local particleParams_red =
31 {
32 flags = { "water", "colorMixing" },
33 linearVelocityX = 256,
34 linearVelocityY = -480,
35 color = { 1, 0, 0.1, 1 },
36 x = display.contentWidth * -0.15,
37 y = 0,
38 lifetime = 8.0,
39 radius = 10,
40 }

```

Using the `physics.newParticleSystem`, we prepare the Box2D physics. Available parameters for `newParticleSystem` are:

- [filename](#) – Required texture file used for an individual particle.
- [baseDir](#) – Folder where the texture files are stored.
- [radius](#) – Radius of the generated particle
- [imageRadius](#) – Allows the image to be rendered with a different radius than the physical body (the `radius` parameter).
- [pressureStrength](#) – Pressure is increased in response to compression.
- [dampingStrength](#) – Slows the velocity of the particle. Larger numbers have more of an impact.
- [elasticStrength](#) – How quickly the particle returns to its original shape.
- [springStrength](#) – How quickly a spring particle returns to its original length.
- [viscousStrength](#) – Slows the velocity of viscous particles.
- [surfaceTensionPressureStrength](#) – Produces pressure strength on tensile particles (i.e. larger numbers give greater surface tension). Range: 0 to 0.2
- [surfaceTensionNormalStrength](#) – A smoothing effect for the outline of the particles; gives a more fluid appearance. Larger numbers result in more water-like appearance. Range: 0 to 0.2
- [repulsiveStrength](#) – Increases pressure on repulsive particles. Negative numbers result in attraction. Suggested Range: -0.2 to 2.0.
- [powderStrength](#) – Produces repulsion between powder particles.
- [EjectionStrength](#) – Particle push from solid particle groups. Larger numbers result in more particles being pushed out of group.

- staticPressureStrength – Produces static pressure on neighboring particles.
- staicPressureRelaxation – Decreases instability of particles caused by static pressure.
- staticPressureIterations – Changes the number of times the static e pressure is calculated for static pressure. Default: 8.
- colorMixingStrength – Set the speed on how quickly colors will mix. A value of 1.0 will result in immediate mix, while 0.5 is 50% mixture per simulation step.
- destroyByAge – Boolean variable that allows particle destruction by age when no more particles are created.
- lifetimeGranularity –Sets granularity of particle lifetime in seconds.
- blendEquation – How the blending equation will occur with particles: Values: “subtract”, “reverseSubtract”, or “disabled”.
- blendMode – Set the particle blending mode: Values: “add”, “multiply”, “screen”, or “normal”.

## A Matter of Perspective

2 ½D, isometric, mode 7, and 3D are all just a matter of perspective when developing games. While it was possible to do isometric or 2.5D on Corona previously, the update to the graphics engine makes it much easier to produce advanced effects and even simulate 3D game play on mobile devices.

One of the most common methods to simulate depth in a game is the use of parallax scrolling. The method has been around since the early days of console games and is now commonly used in side scroller games to give the feeling that the game is taking place in a much larger environment.

## Project 18.5 Side Scroller: Parallax Scrolling

Let's begin by defining parallax scrolling. Parallax has many definitions depending on the application. For our situation in creating a mobile side-scrolling game, it is used to give a sense of depth or distance between the camera and the moving objects as they recede into the distance. The method has been around since the early days of game development and is still commonly used in most games.

The parallax scrolling effect is created by using layers. The back layer, or the one farthest from the camera (usually a distant shot of a sun setting, cityscape, or some type of landscape shot), one or more middle distance objects such as closer buildings, and then close distance which contains our primary character and the immediate obstacles to be overcome.

Thus, for our game of a runner, I will use a landscape for the distant shot, intermittent shots for middle distance such as a house, barn or trees, and then a closer shot of fence or cornfields. Finally, we must have a foreground area that will serve as the reference for the parallax scrolling. Each of these distance settings will have a ratio set for it so that we can calculate how much they should move as the foreground moves. You must always have the front most scene set to 1, else you don't have anything to base the ratio off.

Let's start by setting up the config.lua and build.settings files:

### **build.settings**

```
1 -- build.settings for project: SideScroller
2
3
4
5 settings =
6 {
7     orientation =
8     {
9         default ="landscapeRight",
10        supported =
11        {
12            "landscapeLeft","landscapeRight"
13        },
14    },
15
16 }
```

Since the game is intended to be played as a sidescroller, we will restrict the orientation to landscapeLeft and landscapeRight.

### **config.lua**

```
1 -- config.lua for project: SideScroller
2
3 application = {
4     content = {
5         width = 480,
6         height = 600,
7         scale = "letterBox",
8         fps = 30,
9
10        imageSuffix = {
11            ["@2x"] = 2,
12        },
13    },
14 }
```

We are using a standard config.lua for this project. On to main.lua! I want to use composer for scene control for this game. So our main.lua file just needs to perform some basic setup and call the splash screen:

### main.lua

```
1 -- Project: SideScroller
2 -- Description: Simple side scrolling game. Chapter 18 of Learning Mobile Application &
   Game Development with Corona.
3 --
4 --
5 --
6 display.setStatusBar( display.HiddenStatusBar )
7
8 local composer = require("composer")
9
10 composer.gotoScene( "splash" )
```

We will keep the splash screen fairly simple for the moment. Just setup composer and pre-declare the splash variable. In the **create** function we will create a rectangle the size of the screen and set it as a black background. Then create a simple text message and place it in the center of the screen.

We will also setup the nextScene function so that when the background is tapped, we will remove the text and load the next scene.

### splash.lua

```
1 -- splash
2
3 local composer = require("composer")
4 local scene = composer.newScene()
5 local splash
6
7 function scene:create(event)
8     splash = display.newRect(0,0, display.contentWidth, display.contentHeight)
9     splash:setFillColor(0,0,0)
10
11    local myText = display.newText("Welcome to my Awesome Level",0,0,nil,24)
12    myText:setFillColor(1, 1, 1)
13    myText.x = display.contentCenterX
14    myText.y = display.contentCenterY
15
16    local function nextScene()
```

```

17     composer.gotoScene("1Level")
18 end
19
20 splash:addEventListener("tap", nextScene)
21 end

```

For the remainder of the splash.lua file, it is what you would expect to see in a composer based app.

```

22
23 function scene:show( event )
24 end
25
26 function scene:hide(event)
27 end
28
29 scene:addEventListener( "create", scene )
30 scene:addEventListener( "show", scene )
31 scene:addEventListener( "hide", scene )
32 return scene

```

## 1Level.lua

From here forward, we will be working with the 1Level.lua file. This is where we will configure the parallax as well as basic game play. We will start by calling composer, setup our scenes, and get physics started.

```

1 -- Side Scroller
2 -- Graphics 2.0 parallax scrolling
3 local composer = require("composer")
4 local scene = composer.newScene()
5 local physics = require("physics")
6 physics.start()
7 physics.setGravity( 0, 9.8 )
8 --physics.setDrawMode("hybrid")
9

```

Next, we will setup a few variables that are used throughout the level.

```

10 function scene:create(event)
11     local centerX = display.contentCenterX
12     local centerY = display.contentCenterY
13     local _W = display.contentWidth

```

```

14 local _H = display.contentHeight
15
16 -- Define reference points locations anchor points
17 local TOP_REF = 0
18 local BOTTOM_REF = 1
19 local LEFT_REF = 0
20 local RIGHT_REF = 1
21 local CENTER_REF = 0.5
22
23 local baseline = 380
24 local obj = {} -- used to store parallax objects and obstacles
25

```

Notice that we simplified keeping track of the anchor points by referencing where they set the anchor point, just to make our lives a little easier. In the last two line (23 & 24), we set the base of the parallax scrolling (approximately where I want the action to occur) and setup a table to hold the objects that will be scrolling across the screen. Time to load some of those objects! First, let's get the background sky loaded and setup some text to display how far our runner has run.

```

26 -- load the graphics
27 -- Load the sky and background image. These will not move.
28 local sky = display.newImageRect( "sky.png", 1000, 480 )
29 sky.x = centerX
30 sky.y = centerY
31 local distance = 0
32 local distanceText = display.newText("Distance: "..distance, 100, 40, nil, 24)
33
34 obj[1] = display.newImageRect("background.png", 1200, 167)
35 obj[1].anchorX = LEFT_REF
36 obj[1].x = -130
37 obj[1].y = centerY-35
38 obj[1].dx = 0.01 -- move the background VERY slowly

```

We are also loading our background image that will move as our runner progresses. Nothing should look unusual except the last line (38). Here we have created a variable within our table to help compute how far the background should move with each iteration. I have it set at .01 currently, which means that for every frame, it will move .01 pixels. The game is set to run at 30 frames per second, so the background should move .3 pixels each second, or approximately 1 pixel to the left every 3.3 seconds.

Next we will load the road area. I'm using the same graphic four times so that we can slide it. When a piece of road moves off the screen, we move it to the far right and recycle it. No sense in using more graphic memory than needed!

```
39 -- Road
40 -- This has multiple parts so we can slide it
41 -- When one of the road images slides offscreen, we move it all the way to the right of the
next one.
42 local road = display.newImageRect("road.png", 300, 230)
43 road.anchorX = LEFT_REF
44 road.x = -50
45 road.y = baseline + 20
46 local road2 = display.newImageRect("road.png", 300, 230)
47 road2.anchorX = LEFT_REF
48 road2.x = 250
49 road2.y = baseline + 20
50 local road3 = display.newImageRect("road.png", 300, 230)
51 road3.anchorX = LEFT_REF
52 road3.x = 550
53 road3.y = baseline + 20
54 local road4 = display.newImageRect("road.png", 300, 230)
55 road4.anchorX = LEFT_REF
56 road4.x = 850
57 road4.y = baseline + 20
```

Okay, our road is in place, but since we are using the physics engine, we need something for our runner to run on. We COULD set the road, but I think it will be more efficient to create a simple rectangle in the right place and configure it as a static object. Note: I found in my testing that some devices were not showing the screen the way I wanted due to being extra long. To compensate for this issue, I made the rectangle extra long so that our runner wouldn't fall to their death.

```
58
59 local ground = display.newRect( centerX, baseline+10, _W+250, 10 )
60 ground.alpha = 0
61 physics.addBody( ground, "static" )
```

Time to load our background objects. This is where the parallax really becomes important. I am using 3 objects, each travels at a different speed because of the .dx property (which stands for delta - X, or the change in X position). The value we give dx tells the software how quickly to move the object across the screen. The smaller the number, the further back toward the horizon it is assumed to be. A value of 1 is moving at the same speed as

the runner. Thus, if you wanted to add cars zipping past our poor runner, you could set it at a dx of greater than 1.

```
63 -- Load background objects
64
65
66 obj[2] = display.newImageRect( "barn.png", 204, 200 )
67 obj[2].xScale = 0.6
68 obj[2].yScale = 0.6
69 obj[2].anchorY = BOTTOM_REF
70 obj[2].x = 200
71 obj[2].y = baseline-50
72 obj[2].dx = 0.2
73
74 obj[3] = display.newImageRect( "fence.png", 300, 100 )
75 obj[3].xScale = 0.8
76 obj[3].yScale = 0.8
77 obj[3].anchorY = BOTTOM_REF
78 obj[3].x = 200
79 obj[3].y = baseline -30
80 obj[3].dx = 0.3
81
82 -- insert obstacles
83 obj[4] = display.newImageRect( "trash cans.png", 50, 50 )
84 obj[4].xScale = 1
85 obj[4].yScale = 1
86 obj[4].anchorY = BOTTOM_REF
87 obj[4].x = 20
88 obj[4].y = baseline-10
89 obj[4].dx = 1
90 physics.addBody( obj[4], "dynamic", {density=.5, friction=.5, bounce=.5} )
```

Time to load our runner. I modified the greenman that comes with the Corona Sample code and made him into a blueman (I always did like that group).

```
91
92 -- A sprite sheet with a green dude
93 local sheet = graphics.newImageSheet( "blueman.png", { width=128, height=128,
94 numFrames=15 } )
95
96 -- play 15 frames every 500 ms
97 local blueman = display.newSprite( sheet, { name="man", start=1, count=15, time=500 }
98 )
99 blueman.x = centerX-100
100 blueman.y = baseline
```

```

99 blueman:play()
100 physics.addBody( blueman, "dynamic", {density=2} )
101 blueman.isFixedRotation=true
102

```

## Parallax

Time to get things moving! First, we need the current system time so that we can compute how far the scene needs to move. The first time through, we'll capture the system time in tPrevious so that it's available for later computations.

The function **move** will take the current event time, subtract the previous time and compute the time delta (how much time has elapsed). This is incase something slowed the processing of the scene, or the scene processed faster than normal. Basically, it will keep everything flowing smoothly across the screen. Once we have tDelta, we can also compute the distance. Since our runner is taking a full step every 500 milliseconds, I decided that each time a full second has elapsed, the runner has moved 1 meter forward (which means he has a really short stride). This is used to display the distance traveled for text.

```

104 -- setup parallax scrolling. Layers must be in back to front order with a distance ratio.
105 -- A per-frame event to move the elements
106 local tPrevious = system.getTimer()
107 local function move(event)
108     local tDelta = event.time - tPrevious
109
110     tPrevious = event.time
111     distance = distance + (tDelta*.01)
112
113     distanceText.text = "Distance: "..math.round(distance).. " meters"
114
115     local xOffset = ( 0.2 * tDelta )
116
117     road.x = road.x - xOffset
118     road2.x = road2.x - xOffset
119     road3.x = road3.x - xOffset
120     road4.x = road4.x - xOffset
121
122     if (road.x + road.width) < -120 then
123         road:translate( 1200, 0 )
124     end
125     if (road2.x + road2.width) < -120 then
126         road2:translate( 1200, 0 )
127     end
128     if (road3.x + road3.width) < -120 then
129         road3:translate( 1200, 0 )
130     end

```

```

131 if (road4.x + road4.width) < -120 then
132     road4:translate( 1200, 0 )
133 end

```

After we compute the distance, we next move the road by just changing the road.x value. Remember, we are only moving a fraction of a pixel per frame, so the movement will look smooth.

After we move the road, we check to see if the road has moved far enough off the screen to move it back to the right. Remember, we want the road all the way off the screen before we move it. If it's not, it will leave a gap in the road on the screen for a few seconds, which is not what we want!

Now it's time to take care of the parallax movement. Using a for loop, we can cycle through all of the objects nice and quick, moving them just like we move the road. We check to see if they are off the screen to the left, and if they are, move them to the far right (off screen).

```

134
135 local i
136 for i = 1, #obj, 1 do
137     obj[i].x = obj[i].x - obj[i].dx * tDelta * 0.2
138     if (obj[i].x + obj[i].contentWidth) < 0 then
139         obj[i]:translate( display.contentWidth + obj[i].width*2, 0 )
140     end
141 end
149 end

```

Now we just need to take care of the event listener and composer.

```

160 -- listeners
161
162
163
164 Runtime:addEventListener( "enterFrame", move );
165
166
167
168 end
169
170 function scene:show( event )
171
172
173 end
174
175 function scene:hide(event)

```

```

176
177 end
178
179 scene:addEventListener( "create", scene )
180 scene:addEventListener( "show", scene )
181 scene:addEventListener( "hide", scene )
182 return scene

```

With the addition of the Runtime event listener, you should be able to now run the game.

But wait! There's more! Won't it be nice if our runner could jump over the trash cans? That would make it a little more entertaining. Let's add a function that will do just that. Insert at line 151 a function that will apply a linear impulse to the runner.

### Jumping to Conclusions

```

151 -- handle the controls (tap to jump)
152 local function jump()
153   print("JUMP!!")
154   blueman:applyLinearImpulse( 0, -200, blueman.x, blueman.y )
155 end
156

```

And don't forget a tap area and listener to call the jump command:

```

161 local tapArea = display.newRect( centerX, centerY, _W+255, _H+255 )
162 tapArea.alpha = .01
163
164 Runtime:addEventListener( "enterFrame", move );
165 tapArea:addEventListener( "tap", jump )

```

### Fini (maybe)!

Now we're running! We will add one more situation before calling this game done. Should the runner fail to jump the trash cans and be pushed all the way to the left of the screen, we'll consider the game over. Insert these lines back in our **move** function:

```

142   -- Finished
143   if blueman.x <= -100 then
144     blueman:pause()
145     physics.stop()
146     Runtime:removeEventListener( "enterFrame", move )
147     local finished = display.newText("You ran "..distance.." meters", centerX, centerY,
148     nil, 30)
149   end

```

And there we have it, a simple side scrolling endless runner game!

## Summary

If you have ever wondered about the development process that I use when making a game like this, I wanted to assure you I do not start at the top line and write to the bottom. The process I use is to develop one section at a time, such as parallax, then controls, then sprites, etc. During that process I am adding variables to the beginning of the main file, listeners to the end, and updating previously written sections when something goes wrong. All together, to get this game written from concept to working version for the book took about 2 weeks of part-time work (I am a professor by day; they expect me to show up and teach my classes). I should also mention that those 2 weeks of part-time work draw upon over 30 years of programming experience. In other words, if it takes a while to make your game or app, don't be surprised! Making quality apps takes time. But every app you make will contribute to the next app.

## Questions

- 1) Define parallax. How can it be used in game development?
- 2) What is "Mode 7"? Name at least one game that used Mode 7.
- 3) Describe the difference between a generated effect, a composite, and an effect.
- 4) When should a container be used instead of a display group?
- 5) What does Liquid Fun allow you to simulate?

## Assignments

- 1) Add additional obstacles that must be avoided in Project 18.5: Side Scroller
- 2) Add additional particle flows to Project 18.4: Liquid Fun
- 3) Select a composite and apply it to a picture of your choice.
- 4) Load an image and apply the checkerboard generator to it.
- 5) Add additional layers to the parallax in Project 18.5: Side Scroller. How does the size of the object impact the perception of distance?

# Chapter 19 Native Application Development

## Learning Objectives

There are times when cross-platform tools like Corona are not best suited for your project. In those situations you are left with two choices: native or HTML. In this chapter, we will introduce how to develop a native app using the Google Android APK.

Specifically in this chapter we will:

- Examine the choice to use a native tool set
- Discuss options for native development
- Introduce native app development with Android APK

## Why Android?

As we discussed in Chapter 1, there are many choices in making native apps. The reason that we will be covering Android instead of iOS or Windows comes down to two factors: cost and market share.

### Cost

Google's Android APK is free to download and use for development.

### Market Share

While Apple's iTune store is still the best place to make money with your apps, the Android OS is now the most popular mobile operating system.

## Android Studio

Android Studio is the free IDE for developing Android based apps. It includes all of the tools to test and debug your app. It can be downloaded from

<http://developer.android.com/sdk/installing/studio.html>. While it is a fairly large download, it does greatly simplify developing for the Android platform. If you do not have a recent install of the Java Developers Kit (JDK), I recommend downloading and installing it before you install the Android Studio. Additional comments and instructions are available in Appendix C.

Unlike Corona SDK, all android apps are written in Java. You will find the syntax and requirements different from Lua. But all of the programming skills in developing functions, loops, if..then statements and all the rest carries over. Once you get used to the differences I'm sure you will be creating native apps in no time at all.

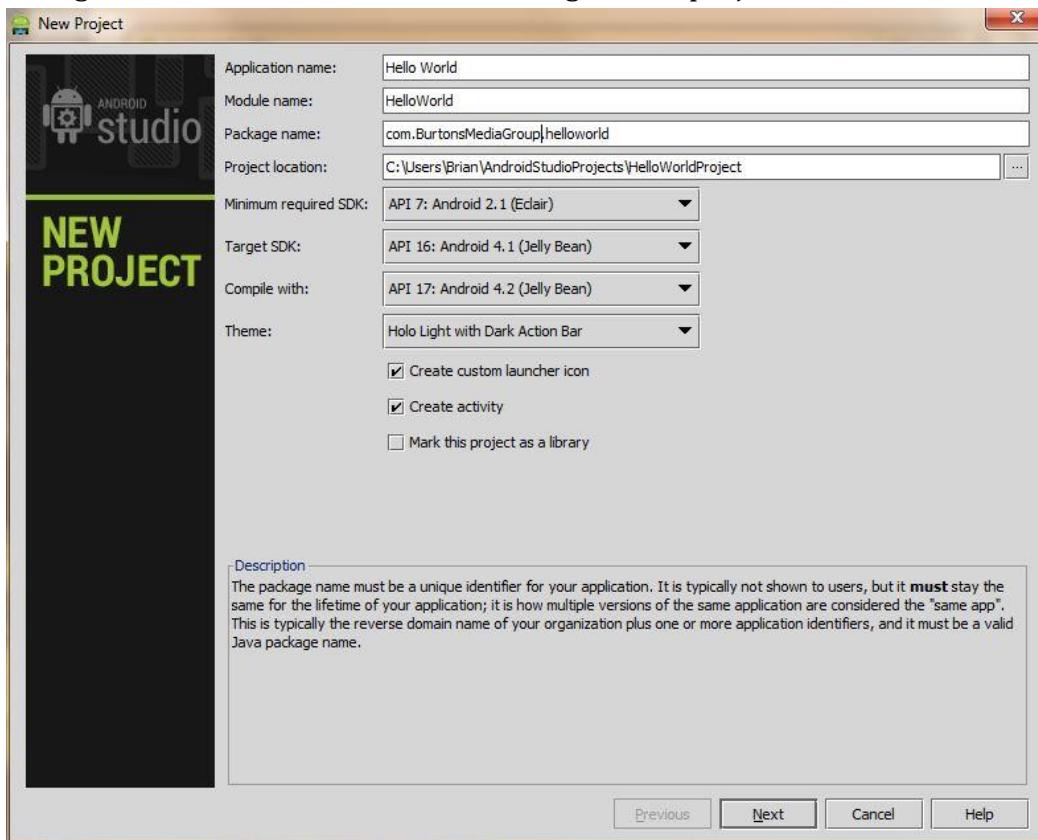
## Project 19.0: Native Android-Hello World

We are going to do a simple Hello World with Android Studio to become familiar with how the IDE works and some of the differences from Lua. To get started, launch Android Studio.



If you have not performed a “Check for updates” (located at the bottom of the startup screen), I recommend you do so before going any further. Go ahead, I’ll wait.

Okay, now that we are up-to-date, click on New Project. This will open the New Project dialog screen which will allow us to configure our project.

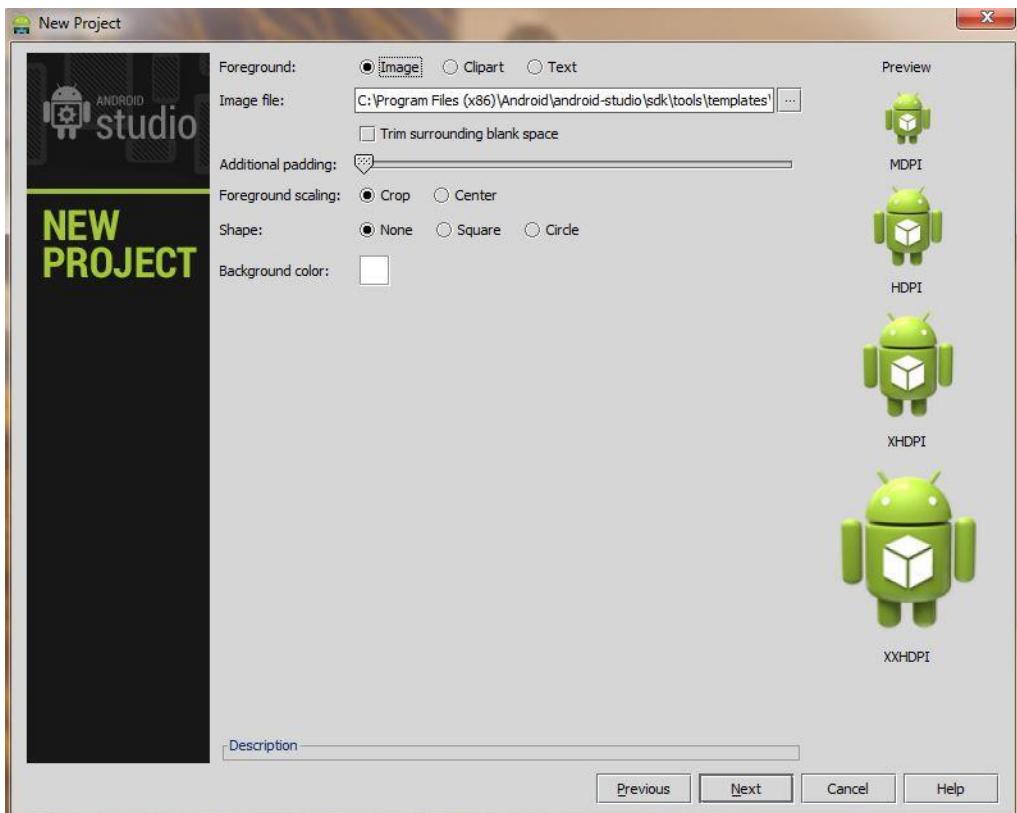


In Application name, enter “Hello World”. As you type, you will notice that Module name, Package name and Project location are all updated for the name of your project.

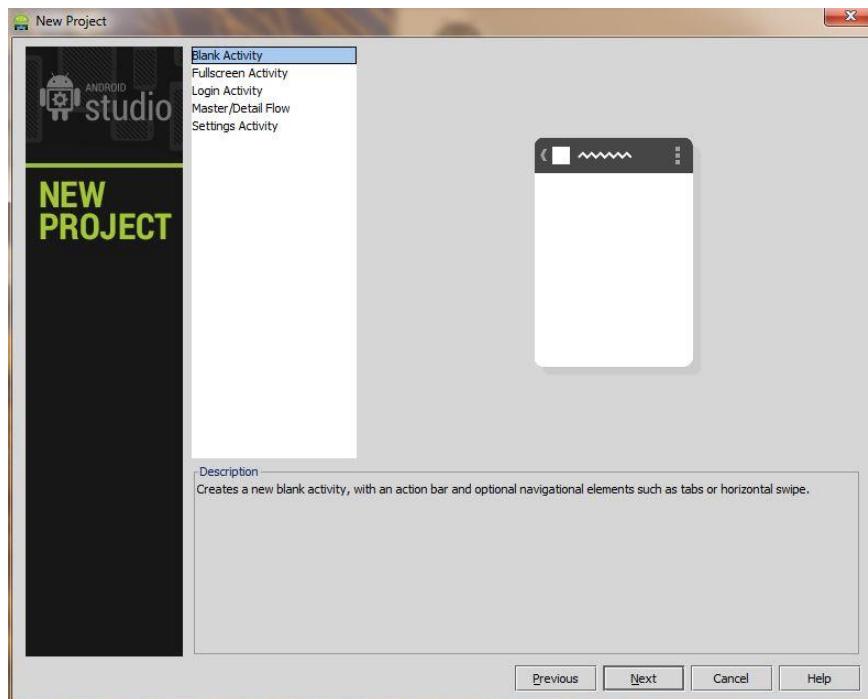
In the Package name, you will want to enter in reverse URL notation your business website, like we did back in Chapter 4 when we discussed publishing to Android devices. This is optional at this point, but something to remember when you are preparing your own apps for publication.

The rest of the options are fine for this project, so click Next to proceed.

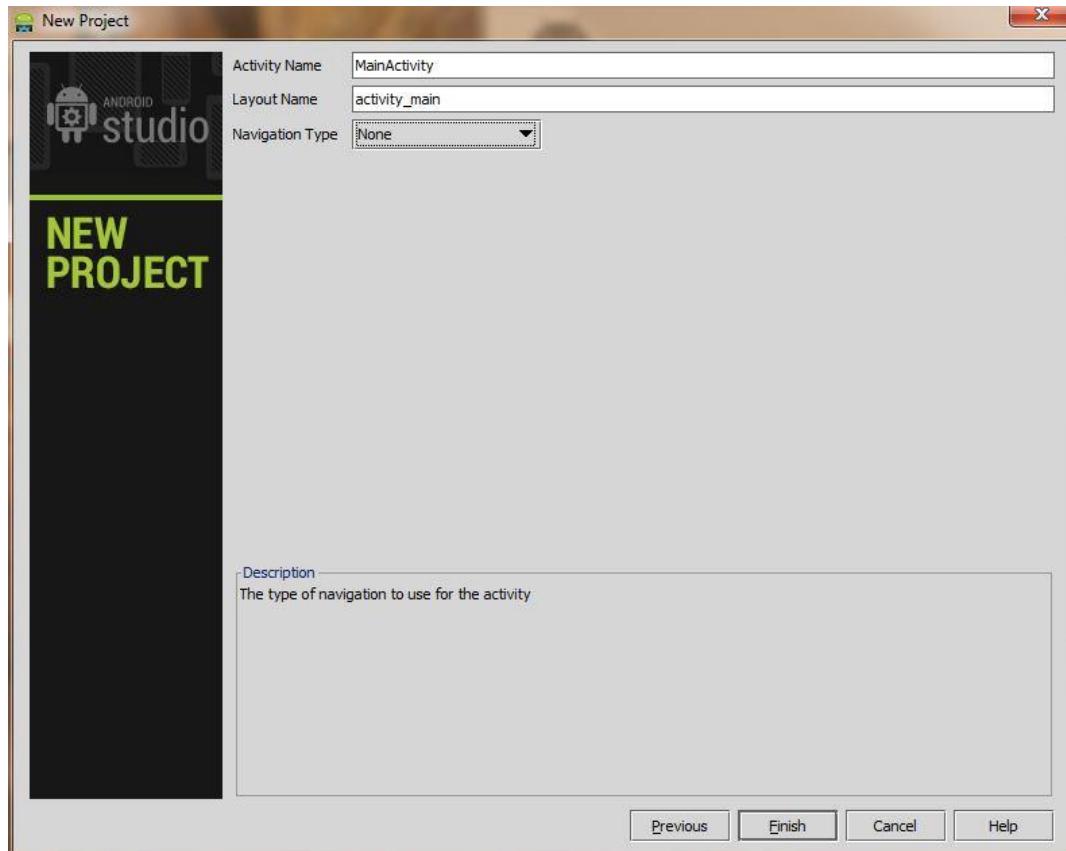
The next dialog screen is focused on setting up the icons for the app and default template. For our “Hello World” project, we can use the default template to keep every thing relatively simple.



The next screen allows you to setup the Android programming environment for more advanced types of apps; for instance, using tabs or tables. We can use the Blank Activity option (the default) and proceed to the next screen.



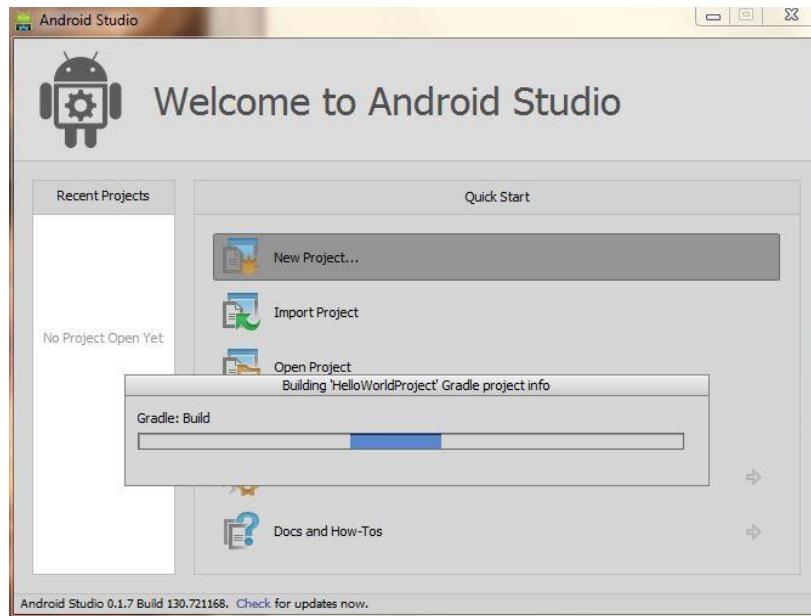
The next screen is for setting up a menu or other type of navigation within our app. Again, much more complex than what we are trying to accomplish with this first app. Just click the Finish button.



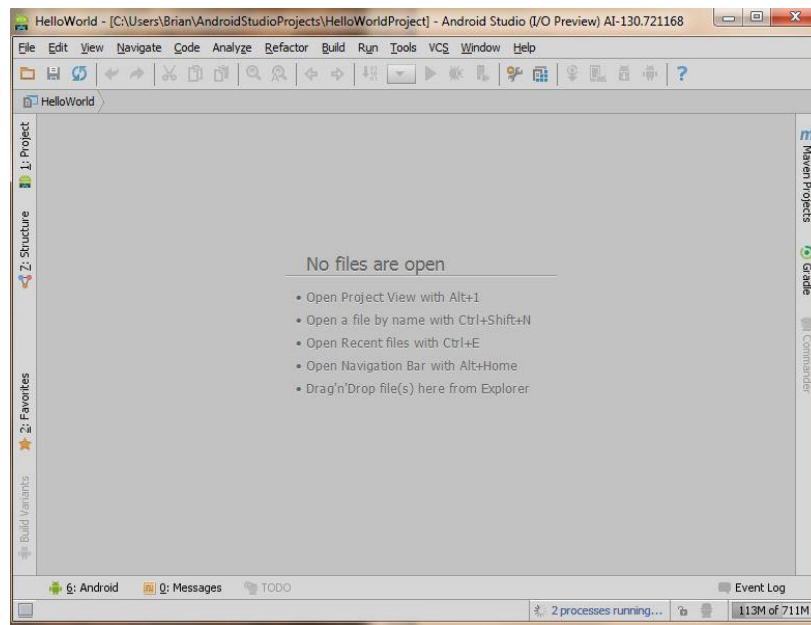
If this is a fresh install of Android Studio in a Windows environment, you might need to Allow Access for the Windows Firewall.



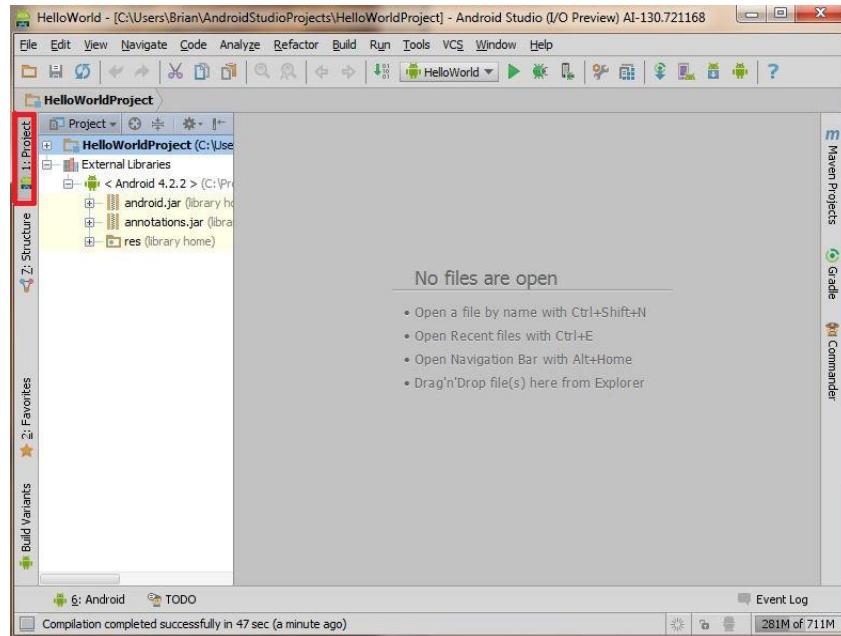
Once you have the configuration set and taken care of any potential firewall problems, you should see Android Studio build the project. This may take several minutes the first time you create a project as Android Studio may need to download some needed Java libraries for the project.



Once the initial project build is completed, you will be greeted by a screen that is strangely blank. To quote the Hitchhikers Guide to the Galaxy: "Don't Panic."



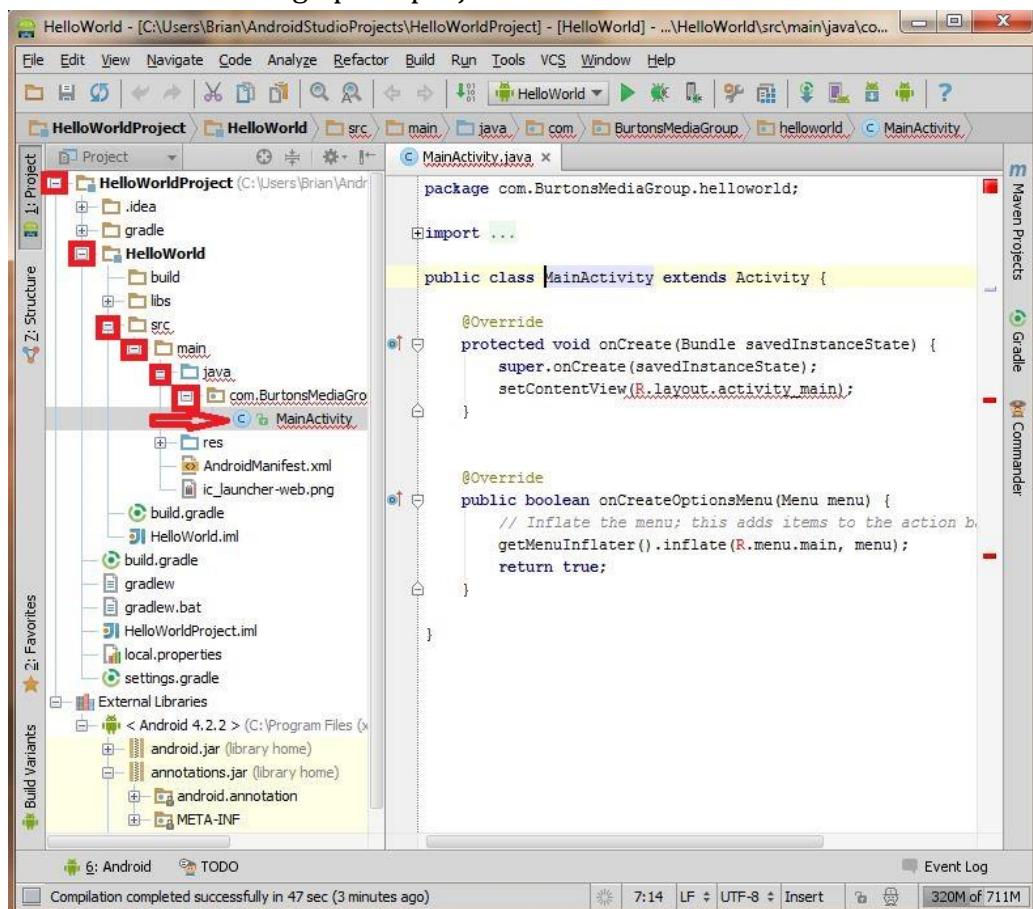
On the left side of the screen you will see the green Android figure with the word “1: Project”. Click on this to open the Hello World Project.



Now we need to find our project work files. Actually, all the files that are being shown are a part of the files to enable Android to work correctly, but we only need to focus on one of those files. Sadly we are going to have to dig for it. Click on the + symbol by HelloWorldProject at the top. This will expand the project ‘tree’. Follow the tree down until you see Hello World again and click on the + symbol beside it. Continue following the tree until you see the src folder. This is short for Source, i.e. the source files from which the program will be built. Click main (main.. sound familiar???) + symbol, then again for java, and com.BurtonsMediaGroup.HelloWorld (or whatever you name your project file in the reverse URL notation).

You should now see a file that is named MainActivity. Click on this file name to open the file. You might remember MainActivity was the name that was in the configuration screens

when we were setting up our project.



Now we are ready to do some programming!

First, a note about Java. All lines of code in Java MUST end with a semi-colon! A line of programming code can span several lines on the screen if need to, but must end with a semi-colon. I have had many a student spend hours trying to get their program to work and all they were missing was a ; at the end of a line.

Similar to Lua, Java allows us to require outside or external files to make our app development easier. In Java, these external files are loaded with the keyword import.

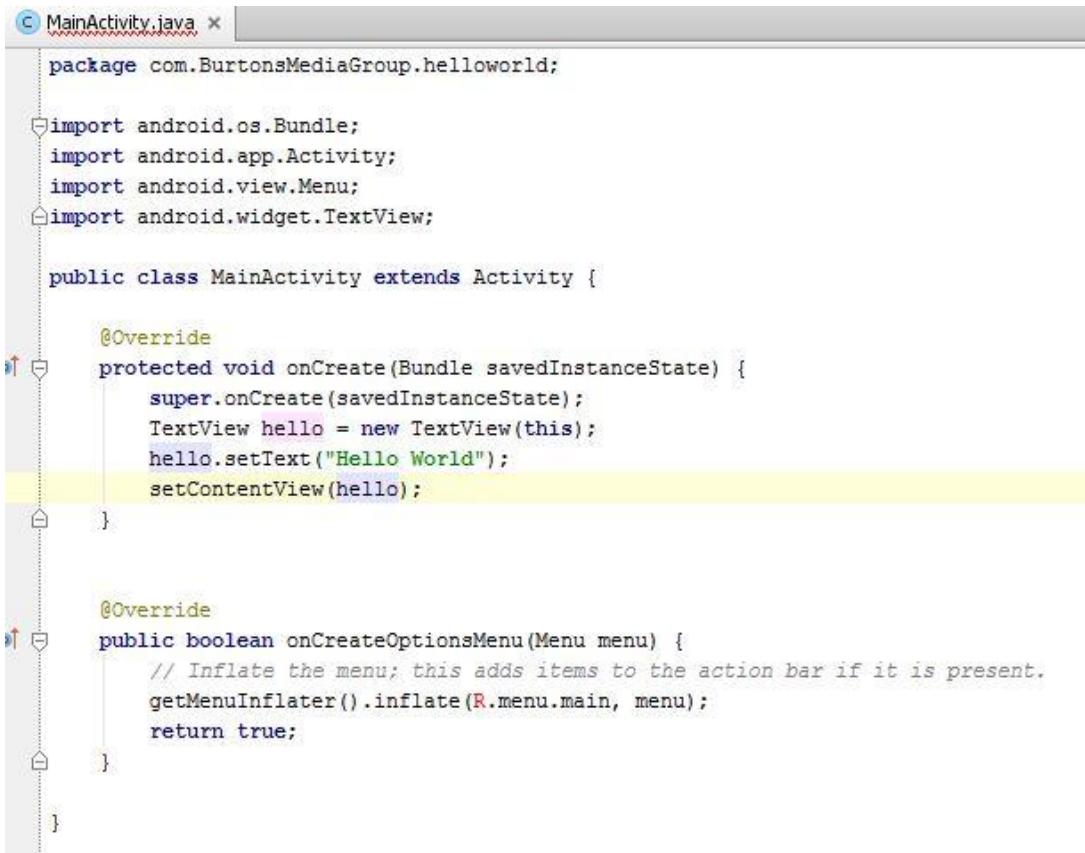
Click on the + symbol beside import to expand all of the imports currently in the project. We need to add an additional import for our text to show on the screen. Below the last import statement, type:

```
import android.widget.TextView;
```

This tells Android that we will be using the text widget in our app. This will enable us to display our “Hello World” message.

Functions (also called methods) are a little different in Java as well. Look down to where you see protected void onCreate....

This is the equivalent to local function onCreate(). Protected is a key word meaning that function can only be accessed from within the app. Void tells the Java compiler that there is nothing returned from this function.



The screenshot shows the code for MainActivity.java in an IDE. The code defines a Main Activity that extends Activity. It overrides the onCreate() method to create a TextView and set its text to "Hello World". It also overrides the onCreateOptionsMenu() method to inflate a menu. The code is as follows:

```
package com.BurtonsMediaGroup.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.TextView;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView hello = new TextView(this);
        hello.setText("Hello World");
        setContentView(hello);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
}
```

Under the line “super.onCreate(savedInstanceState);” type:

```
TextView hello = new TextView(this);
hello.setText("Hello World");
```

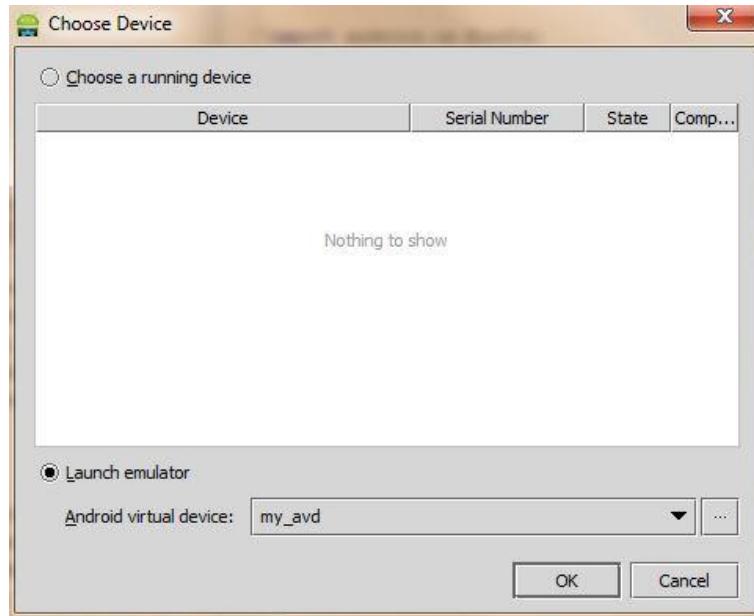
and change the final line to:

```
setContentView(hello);
```

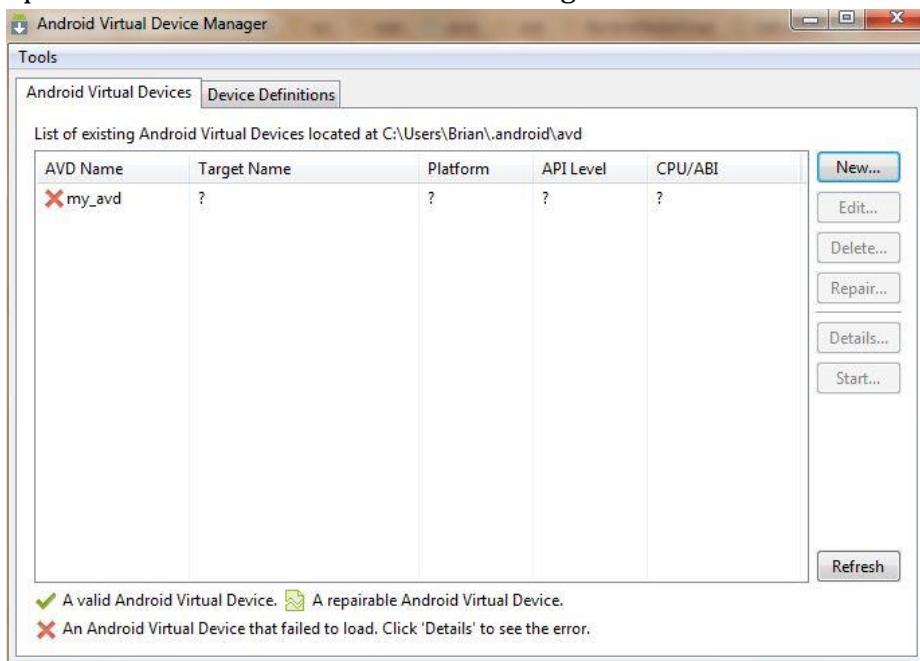
What did you do with these 3 lines of code?

- 1) You created a new TextView object called hello.
- 2) You set the text of hello to “Hello World”
- 3) Set the view (what will be displayed on the screen) to the variable hello.

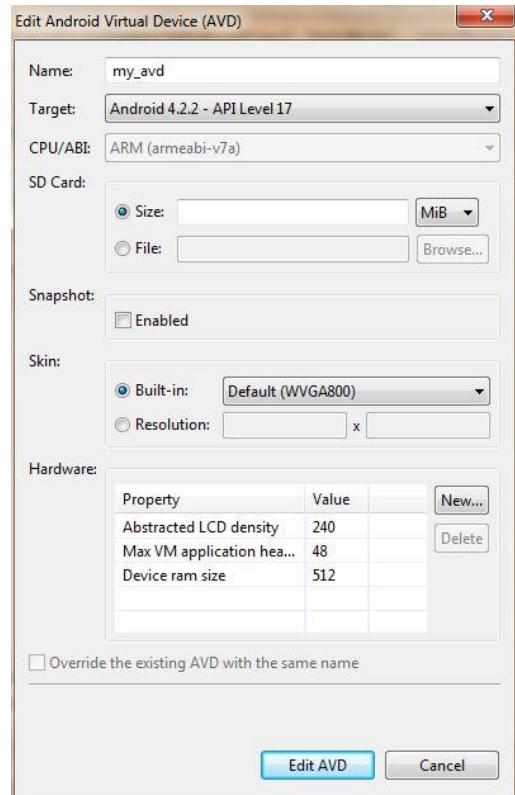
Time to test it! Click the Play button on the tool bar. You will be asked if you want to run this app on a device or to launch the emulator. As you can see, I have the emulator selected.



The first time I tried to launch my app, nothing happened, I had to go back and click the three dots to the right of the my\_avd dropdown box (avd: Android Virtual Device). This opened the Android Virtual Device Manager window.

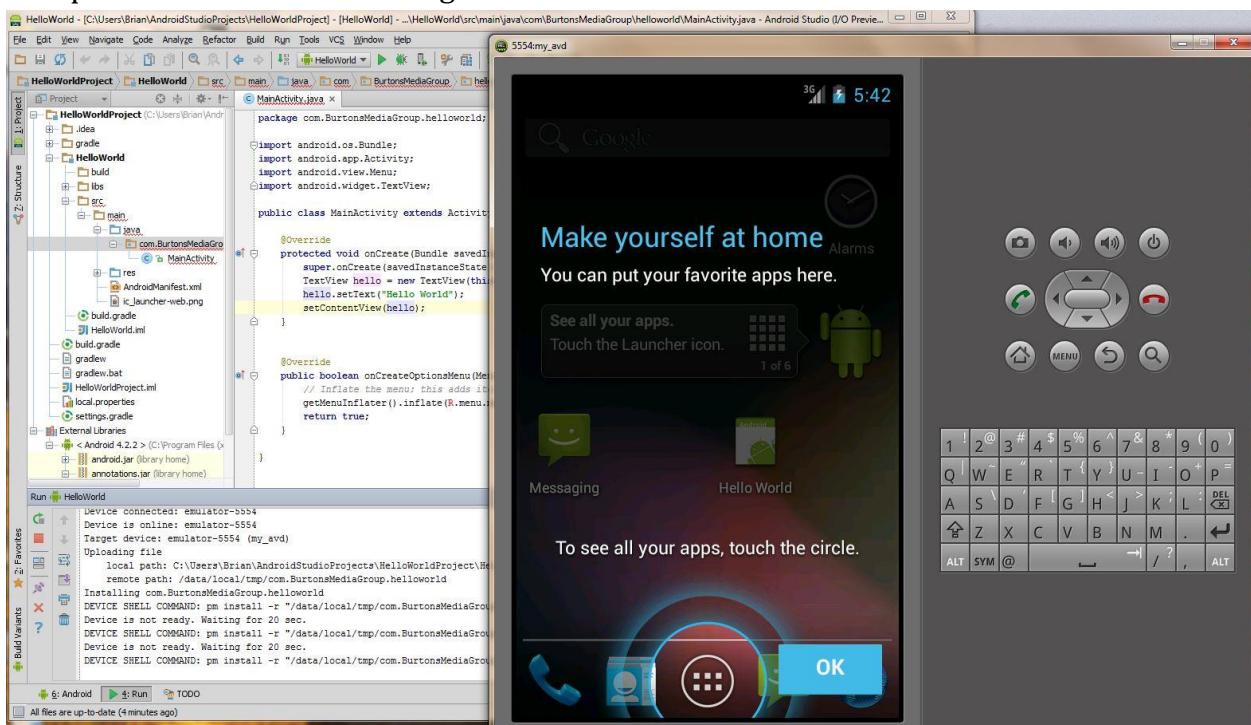


Once I clicked on my \_avd then the edit button, it brought up the device and I was able to select a target device.

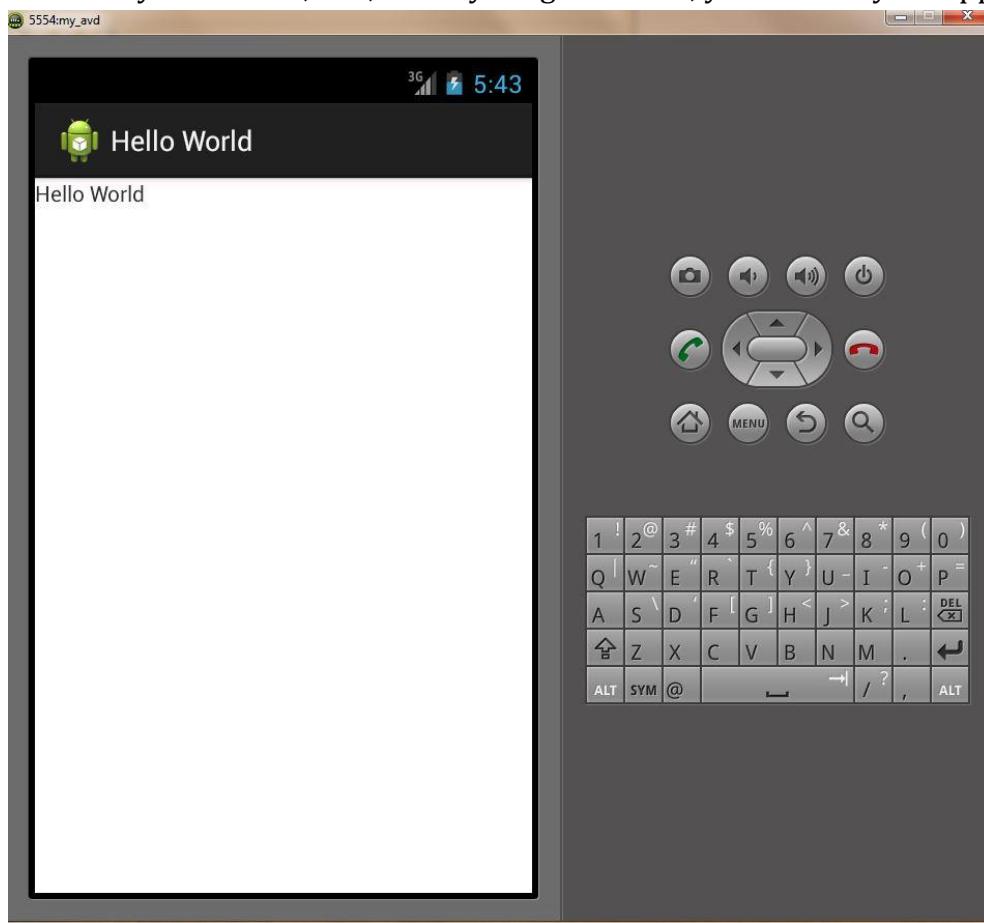


Once that was selected, I just clicked on Edit AVD, and it was off and running. The first time I launched the AVD, it took a LONG time to load, and even longer for the Android OS to load, then it did an update. If you are concerned about the progress, watch the terminal window

for updates as the AVD is loading.



Eventually it will load, and, if everything is correct, you will see your app launch (Yea!).



## Summary

After creating your first native app in Android, you might be wondering if it is similar for iOS or Windows. The answer is a resounding YES. iOS and Windows share a similar complexity. You might remember that we did all of this with 2 lines of code with Corona back in chapter 1 (and the second line was to set the text color).

## Questions

- 1) True or False: Android will only run on Google developed devices.
- 2) True or False: To develop for Android, you must use the C++ programming language.
- 3) True or False: Android is the most widely installed Smart Phone operating System.
- 4) Discuss why you would choose to develop native instead of cross-platform.

## Assignments

- 1) Modify Hello World to give a shout-out to your parents and/or teacher.

- 2) Change the text color of your Hello World.
- 3) Add 3 additional messages to the screen, each in a different color.

# Chapter 20 Next Steps and 3<sup>rd</sup> Party Resources

## Where do we go from here?

I think you will believe me when I tell you that we have only scratched the surface with what you can do with mobile application & game development! There are many great resources that you can use to continue your education. I recommend that you start with the Corona Labs website. Watch the forums, blogs, tutorials and updates. I am sure that they will spark many creative ideas and help you to complete your personal and professional projects.

A new feature that will soon be available (or is already available, depends on when you purchased the textbook) is the graphics 2.0. Graphics 2.0 will allow Corona to implement shaders, 2.5 graphics, and dramatically improved graphic improvements. I have already seen a few demos created with the new graphic tools set, and to say its performance was impressive would be an understatement. Be sure to

## 3<sup>rd</sup> Party Resources

If you haven't notice, one of the strongest features of the Corona SDK is the great community that uses it to develop their apps. Over the past several years a few great resources have become available that you should be aware. I've not attempted to list every resource, but here are a few that might help you with your projects.

Note: all prices are in USD and were accurate at the time of writing. Software is listed in alphabetical order.

### IDE/Editors

Selecting the right editor is much like purchasing a car; you need to find the one that fits your needs and style. There are many editors available for the Corona SDK so you can find the one that fits your needs and style!

#### **Lua Glider IDE(Mac/Win) by M.Y. Developers**

Originally named Cider, LuaGlider is the newest iteration of what is fast becoming an incredible IDE. With built-in Remote (i.e., use your device to remotely pass accelerometer and other device-only actions to the simulator) and sophisticated debugging features, LuaGlider is one handy IDE.

Cost: \$39.99

Website: <http://www.mydevelopersgames.com/Glider/>

**Outlaw (formerly Corona Project Manager)** (Mac/Win) by J.A. Whye.

Outlaw has a built in editor. Coupled with its ability to greatly simplify tracking your Corona projects, the cost of Outlaw is well worth it. Outlaw is the primary tool that I use for my projects.

**Cost:** \$29.99

**Coupon:** Use the code DRBOOK on the second page of the order form to receive a discount. A free lite version is included with your Corona download.

**Website:** <http://outlawgametools.com/>

## Sublime Text 2

The official editor of Corona Labs. Corona Labs recently published a plugin for use with Sublime which can be found at: <http://coronalabs.com/products/editor/>

**Cost:** free trial, \$70.00

**Website:** <http://www.sublimetext.com/>

## TextWrangler by BareBones Software, Inc.

A free text editor that is very popular. First choice by those who are just getting started and are not sure if they want to purchase a full IDE.

**Cost:** free

**Website:** <http://www.barebones.com/products/textwrangler/>

## ZeroBane Debugger (Mac/Win/Linux) by ZeroBane Studio

IDE with built-in debugging tools.

**Cost:** Donation

**Website:** <http://studio.zerobrane.com/>

## Sample Code & Templates

### Glitch Games

Glitch Games has several very useful libraries for to help your projects be completed more quickly. They have created a github repository with their wonderful resources:

**Website:** <https://github.com/GlitchGames>

## Gymbyl

A nice collection of open source code and particle effects. Check Caleb's links under the code tab for more information.

Website: <http://www.gymbyl.com/>

## LearningCorona

The Learning Corona website was one of the first outside resources to help organize tutorials and samples. While some of the links are no longer working, it continues to be a great resource.

Website: <http://www.learningcorona.com/>

## Outlaw Game Tools

J. A. Whye has created a nice series of tutorial and lessons that can help you get a little further down the road with learning Corona. Some are provided free of charge, for others there is a small fee attached.

Website: <http://outlawgametools.com/corona-sdk-tutorials/>

## Roaming Gamer

Ed has a lot of great templates that you can use to quickly complete projects. Some of the templates are free, others have a small fee.

Website: <http://roaminggamer.com/makegames/>

## Graphics/Physics/Level Editors

### Amino (formerly Spriteloq)

Awesome tool and a great time saver for converting flash swf, png, jpg, bmp, tiff or whatever into great, compact sprite sheets! Unfortunately it has become very expensive. Available for Mac or PC.

**Cost:** \$149.00; free 30 day trial

Website: <http://lanica.co/about/animo/>

## Lime

I have been using Lime for quite a while to make tile based games with Corona. It works smoothly with Tiled and makes level design easy. Now that it is an open source project, it

is even useful (because it is free)!

**Cost:** Free/Open Source

**Website:** <http://lime.outlawgametools.com/>

### **Physics Editor by code'n'web – Andreas Löw**

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.

**Cost:** \$19.95

**Website:** <http://www.codeandweb.com/>

### **Spine by Esoteric Software**

Originally a kickstarter campaign, Spine has quickly taken the sprite animation world by storm. With Spine you can easily make sophisticated animations that can then be exported for use with Corona SDK or a number of other game development environments. Available for Mac or PC.

**Cost:** \$55

**Website:** <http://esotericsoftware.com/>

### **SpriteHelper & 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 developing. 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:** Vladu Bogdan

**Cost:** SpriteHelper: \$16.99; LevelHelper: \$24.99

**Website:** <http://www.gamedevhelper.com>

### **Texture Packer by code'n'web – Andreas Löw**

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.

**Cost:** Pro - \$29.95

**Website:** <http://www.codeandweb.com/>

## Graphics

While there are many professional tools available for creating graphics, here are a few really great free tools to help the Indie get started.

### GIMP

One of the great open source successes, GIMP is a popular bitmap tool for developing and editing graphics files. It is fast, easy to use, and best of all FREE. Available for Windows or Mac.

**Website:** <http://www.gimp.org/>

### Inkscape

Another great open source tool is Inkscape. While GIMP focuses on great bitmap resources, Inkscape is a free tool that allows you to create scalable vector based graphics. In other words, great graphics that scale well. Available for Linux, Mac, and Windows./

**Website:** <http://inkscape.org/>

### Blender

While you might not have much need for Blender when working on sprites or graphics for mobile apps, you should be aware of a great open source tool for making 3D models. Blender is a popular free resource and could be used for creating complex sprites.

**Website:** <http://blender.org>

## Audio

I have included a few popular audio editing/loop – creation tools:

### Acid Music Studio by Sony

One of the first loop editing tools available that regularly provides free sound loops for download. I've kept my license up-dated!

**Cost:** \$64.95

**Website:** <http://www.sonycreativesoftware.com/acidsoftware>

## Bfxr

Bfxr is a great tool for generating royalty free sound effects. Based on the popular SFXR, BFXR goes the next step to quickly generate a multitude of sound effects that you can freely include in your game or app.

**Cost:** free

**Website:** <http://www.bfxr.net/>

## GarageBand by Apple. Mac Only.

Easy to use and you can create loops on your iPad or iPhone! Very popular amongst macophiles!

**Cost:** \$14.99

**Website:** Mac or iTunes store.

## Ocenaudio by OcenAudio. Linux/Mac/Win

I heard about ocenaudio through a friend that swears by it as the tool for creating their audio.

**Cost:** free

**Website:** <http://www.ocenaudio.com.br/>

## App Testing

There are a few services available to simplify testing of your mobile apps.

### HockeyKit

Similar to testflight, but is an open source project under the MIT license. Requires a little more technical know-how on the developers part, but also gives you more control.

**Cost:** free

**Website:** <http://www.hockeykit.net>

### TestFlight

Provides testing services so that you can easily distribute your app without going through the Amazon/Apple/Google/Samsung/etc. app store approval process.

**Cost:** free

**Website:** <http://www.testflightapp.com>

## **Mockup/Design Tools**

Being able to quickly and easily design and develop your app flow is critical. Here are a few commonly used tools for designing the look and flow of your app.

### **BluePrint**

BluePrint is an app for your iPad. It allows you to quickly and easily create a mockup of your apps. It allows the insertion of widgets, customization and free hand sketching on your iPad. You can then share the mockup with other team members or customers so that you have an agreed upon app design.

Cost: free

Website: <http://www.groosoft.com/blueprint/> | iTunes App Store.

### **iPhoneMockup**

An easy to use website for featuring drag and drop tools for creating quick mocks for the iPhone. Easy to use and straight forward.

Cost: free

Website: <http://iphonemockup.lkmc.ch/>

### **Mockflow**

Several professional app developers that I know like and use Mockflow for predesigning their apps or webpages. Mockflow allows you to work collaboratively on complex projects to decide on the right flow for your project. Web or desktop based, it helps you to share the design concept with everyone in your team or your clients.

Cost: Free trial/\$69 yr.

Website: [www.mockflow.com/](http://www.mockflow.com/)

## Appendix A: Installation of Corona SDK

If you are not ready to purchase Corona, you can get started with the unlimited (i.e. it won't time out on you) trial version of Corona.

Installation of Corona SDK is a straightforward project. Just click on the download button at <http://www.CoronaLabs.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).

### CoronaSDK 2012.840

Select your operating system:



#### Corona SDK on Windows Build 2012.840



Corona SDK-2012.840.msi (62.9 MB)

md5: deb362453128a17bc3e2a543eb3a3bc4

Includes Corona SDK Simulator, sample apps, and free 30 day trial versions of Corona Project Manager, Kwik, and Spriteloq.

Please note: Due to Apple's restrictions, you cannot build for iOS on Windows.

System Requirements:

Windows XP or later, 1 GHZ processor

Device support:

Android OS 2.2 or greater (ARMv7). Corona-built apps will not install on Android ARMv6 devices.

[Getting Started Guide](#) [Release Notes](#)

[Download](#) A large orange button with the word 'Download' and a downward-pointing arrow icon.

*Corona SDK Download Screen*

### 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).

Throughout the book I recommend using the Corona Terminal to launch Corona instead of the simulator. The Corona Terminal can be found through your Finder under Applications > CoronaSDK > Corona Terminal.

If you plan to publish to Android devices, the Java JDK should already be installed on your system. You will need to ensure that your test device is running version 2.2 or higher and ARMv7 processor (ARMv6 is no longer supported).

## Windows

Corona SDK for Windows has low hardware requirements. If you have a system that can run Windows, it can run Corona SDK:

- Windows 8, 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

- Android devices running version 2.2 or higher and ARMv7 processor (ARMv6 is no longer supported)

## Problems?

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 not 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 to avoid problems later. 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.

## Appendix B: Publishing to Apple - Xcode & Apple Provisioning

### Xcode

Xcode is now available through the Macintosh App store. It is free if you are a developer; \$5 if not. Installation of Xcode is a standard application installation now that it is a part of the app store and doesn't require any special configuration on your part.

When used with Corona, Xcode does simplify installation of apps to your test device. Simply use the Organizer (available through the menu Window > Organizer). The Organizer allows you to configure your iPad or iPhone as a developer device, install certificates and install apps very easily.

### Types of Apple Developers

Apple offers 3 programs for developers:

- Standard Developer – The most widely used. For publishing apps and games to the iPhone/iPad App store.
- Enterprise Developer – For building enterprise apps for an organization. Is only for apps that will be used internal to the organization. Does not permit you or your organization to sell apps on the apps store.
- University Developer – Used by colleges and universities for training. Allows everything but selling apps or deploying to large numbers of devices. If you attend a college or university that has a University Developers license, this is a great way to get your app ready and thoroughly test it before purchasing your standard license.

For most developers (including large businesses) you will want a Standard Developer. It currently costs \$99/year, so unless you have money to burn, the general recommendation is to not sign-up until you are ready to publish your app.

### Apple Provisioning Profiles and Certificates

The Apple provisioning process is a constantly evolving process. Fortunately Apple provides great tutorials on the developer's site. Simply login with your developers account to <http://developer.apple.com>. You will be able to follow the directions to setup your provisioning profile and app certificates.

Apple requires that you be a standard developer to publish to the Apple iTunes store. You must have a provisioning profile created for the app. To submit your app to Apple, you must build for distribution.

## Icons

Apple requires appropriate icon sizes for the type of device you will be deploying to (57x57, 72x72, 114x114, and/or 144x144) as well as the app icon at 512x512 (though 1024x1024 is recommended) for the store.

## Corona SDK & the App Store

You may have noticed on some of your Corona builds for a device, that Corona will tell you if the app meets the Apple app store requirements. This only looks at the technical requirements of Apple's app store. Take advantage of the feedback from Corona to make your app technically compliant. This does not guarantee that the app will be accepted by Apple.

# Appendix C: Publishing to Android - Installation and Configuring Keystores

## Android Studio Installation

In Chapter 19, we introduce creating native mobile apps with Android Studio. To use Android Studio you will need to install a 64-bit version of Java JDK and the Android Studio. Java JDK can be downloaded from

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>. You will only need the JDK (Java Developers Kit) for your operating system. You will also need to download and install Android Studio: <http://developer.android.com>. I recommend installing the new JDK before the Android Studio. If the JDK is properly installed, then Android Studio will be able to find it and configure your system with a minimal of fuss.

Once you launch Android Studio, it is recommended that you run an update immediately to handle any bug fixes or other updates. The update option is at the bottom of the initial Android studio screen.

## Icons

For submission to Android stores, icons should be named: icon-hdpi.png, icon-mdpi.png, icon-ldpi.png

## Keystores

Android requires a valid keystore for submission of your application. This is their method of ensuring that identity of the author for Android apps. As with Apple, this process is constantly evolving. You can find the latest directions and information on signing your apps at <http://developer.android.com/tools/publishing/app-signing.html>.

## Stores

There are MANY stores available to publish your Android app including (and I am sure I'm leaving several out): Amazon, Barnes & Noble, Google Play, Green Throttle, Ouya, and Samsung. Each store has its own requirements and many require that you pay a developer fee to join as a developer. As with Apple, I recommend that you investigate the stores that you plan to sell your app, but not pay any developer fees until you are ready to publish (unless it is a onetime fee).

## Appendix D: The Lua Language

This appendix on the Lua scripting language was supplied by the great people at Corona Labs. While I have performed a little editing clarification, I want to give credit and thanks to Corona Labs for their support and providing this and the following chapter.

### 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. Corona Labs 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:

<code>==</code>	is equal to
<code>~=</code>	is not equal to
<code>&gt;=</code>	is greater or equal
<code>&lt;=</code>	is less or equal
<code>&gt;</code>	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]
fmod	numerator, denominator	Returns the remainder of [numerator] / [denominator]

Function	Parameters	Returns
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 E, we'll cover more advanced topics, including:

- Variable scope
- Tables as arrays and objects
- Closures
- Side effects

# Appendix E: Advanced Lua Language

This appendix on advanced Lua programming was supplied by the great people at Corona Labs.

## 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's 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, lastIdx = 2 }
local str = ""
repeat
    idx = next ( myTbl, idx )
    if idx ~= Nil then
        str = str .. ", " .. myTbl[idx]
    end
until not idx
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, lastIdx = 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

Operator	Metamethod Signature	Description
*	<code>__mul( a, b )</code>	Called when the parent table is multiplied by a value using the * operator
/	<code>__div( a, b )</code>	Called when the parent table is divided by a value using the / operator
%	<code>__mod( a, b )</code>	Called when a value is used as a modulus of the parent table using the % operator
^	<code>__pow( a, b )</code>	Called when the parent table is raised to the power of a value using the ^ operator
-	<code>__unm( a )</code>	Called when the parent table is used with the unary operator -
..	<code>__concat( a, b )</code>	Called when a value is concatenated to the parent table with the .. operator
#	<code>__len( a )</code>	Called when the length of the parent table is queried with the # operator
==	<code>__eq( a, b )</code>	Called when the parent table is compared with a table also registered with the same metamethod
<	<code>__lt( a, b )</code>	Called when the parent table is compared to a value using the < operator
<=	<code>__le( a, b )</code>	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:

$$c = \text{__add}( a, b )$$

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
```

```

        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.

## Appendix F: The Path to Certification

### What is a Corona Certified Developer?

A Corona Certified Developer is someone who has distinguished themselves in the development of cross-platform mobile applications. They have successfully published applications developed with the Corona SDK. If you have worked through the entirety of this book, you have been introduced to every concept that is covered on the Corona Certified Developer exam.

### Why become certified?

The Corona Certified Developer (CCD) is able to advertise their app development services with a certified developer; showing that they are experienced and knowledgeable in using the Corona SDK to develop applications.

### CCD Exam Structure

The CCD Exam is divided into 7 topics: Basics, Media, Events, Data Storage, Native, User Interface, and Distribution. The exam includes multiple choice, true/false, and fill in the blank questions. Many times you will be asked to determine the output or result of a section of Corona script. Generally you should expect a minimum of two questions per subcategory. At the time of this writing there are 34 subcategories included in the exam.

The CCD exam is constantly evolving as additional questions are added. As new API commands and features are added to Corona, the certification exam will reflect those changes. I have made every effort during the writing of this textbook to ensure that all of the topics in the CCD have been studied. If you have been paying close attention, you should be ready for the exam with just a review of the study outline (included below).

### How Much Does Certification Cost?

The CCD exam costs \$99.00. This allows you to take the test twice (should you not pass it the first time). Once you have passed the exam (how long the certification is good for), you must maintain your Corona SDK subscription to be considered an active certified developer.

### How to Pass the CCD Exam

The CCD exam is designed to test the breadth of your knowledge in using the Corona SDK for app development. A 70% is required to pass the exam. Additionally, you must have

developed an app that is available on one of the app stores (Amazon, Apple, Barnes and Noble, or Google). A brief introduction to all of the topics on the CCD exam is included in this document. If you wish more in-depth coverage, Dr. Brian Burton has created additional training materials and an online certification course that are available through his website: <http://www.BurtonsMediaGroup.com/books> and <http://www.BurtonsMediaGroup.com/classes>. Dr. Burton is also available for on-site group or business training.

## **CCGD Exam**

Yes, the rumors are true. There is a Corona Certified Game Developer (CCGD) (currently in development). The CCGD is considered an add-on to the CCD; i.e., you must pass the CCD to be eligible for the CCGD. The CCGD exam covers the additional topics of physics, monetization, and social networks.

# The Corona Certified Developer Examination & Requirements

The Corona Certified Developer (CCD) program has been developed to recognize those developers who have reached a level of expertise in their app development. The CCD allows others to quickly see which developers are experienced and have a solid understanding of app development with Corona SDK.

The CCD exam is comprised of multiple choice, true false, and short answer questions. You are required to have successfully published an app to an app store and will be asked to provide the app name and store in which it is available during the test.

You will find the exam covering a broad range of topics; basic understanding of developing functions and modules, listening for events, handling media and networking, and data storage and retrieval to name a few. This document will review the concepts that you should be familiar with and prepared to find on the test. To aid in your understanding, we have broken the topics in to seven categories: Basics, Media, Events, Data Storage, Native, User Interface, and Distribution.

This document is designed to provide you with a brief overview of the topics covered by the CCD.

## Section I: The Basics

The Basics are a set of essential skills that all Corona Developers should be familiar:

**API Reference** – The API (Application Programming Interface) and its use from the Corona Labs website.

**Control Structures** – Understand if statements and loops and how they are implemented in Lua.

**Functions** – How to create a function and the different methods of implementing functions. Understand the advantages of the different implementation methods. How to pass information to and from a function.

**Non-image Vector Object**– Understand the difference between a vector image and a bitmap image. When to use a vector and how to create a vector object.

**Debugging** – Basic principles of how to debug an app.

**External Modules** – How to call external modules; best practices in calling modules.

**Use of Custom Fonts** – How to embed and load custom fonts in an app.

**Tables** – How to create a table and when they should be used. How to create and use a multi-dimensional table.

**Strings** – How to use various string functions to change or return information from a string variable.

**Use of Timers** - How to use a timer, including setting up a timer, how to delay an operation, and remove a timer.

**Local vs. Global** – Understand the difference and proper usage of a local and global variable.

## Section II:Media

Different types of media are essential to a good app. Corona offers a variety of resources for media.

**Dynamic Image Resolution** – How to configure and load different resolutions of images into an app.

**Content Scaling** – How to use content scaling within an app.

**Animation** – How to create animated effects within Corona using the 'enterFrame' listener.

**Audio** – The difference between streaming sound and sound that is loaded. Recognize when and how to use the different types of sound/music API commands.

## Section III: Events

Events happen. How you handle those events can be the difference between a successful app and a non-successful app.

**Event Listeners** – How to use event listeners, including listening for different events, passing parameters, and handling system events.

**Touch/Multi-touch** – How to handle touch and multi-touch events within an app.

**setFocus** - How and when to use the setFocus command and the impact this has upon an object.

**Accelerometer, Location, System** – How to use or handle various system events in your app.

**Runtime** – Understand what runtime events are, when they happen, and how to handle them.

**Orientation Change** – How to handle an orientation change event within an app and the various event parameters associated with this event.

## Section IV: Data Storage

The ability to access and store information is critical for the success of an app.

**Working with a Database** - How to read and write to a SQLite database.

**Read/write from a file** – Understand the difference between implicit and explicit file input and output. Should also be familiar with the commands associated with implicit and explicit file I/O.

**JSON** – How to use JSON to send or receive information from a file or other source.

## Section V: Native

Android and iOS offer many native resources. Familiarity with how to use these to your advantage is critical.

**Webviews** – How and when to use webviews in an app. How to create different types of webviews.

**Alerts** – When to use and how to configure an app to handle alerts.

**Text fields and text boxes** – How to use and the difference between text fields and text boxes.

**Async HTTP requests** – Be familiar with the types of requests and how to create a request and to handle the results.

## Section VI: User Interface

Corona offers great tools for building great user interfaces.

**Composer** – How to use the composer API and when to use the various commands associated with composer.

**Widget** - How to configure widgets and their themes. Should be familiar with the types of widgets and their uses.

## Section VII: Distribution

You've made a great app thanks to Corona, but how do you get it into the stores? CCD must have published at least one app to an app store.

**Build for the app stores** - Familiarity with the basic requirements of the Apple and Android stores.

**Acquire certificates and digitally sign apps** - Familiarity with the Apple provisioning process and Android Keystore signing requirements.

**Pass Apple or Amazon's app review process** - This is a fill-in-the-blank question where you will provide the name of an app and the store that it is available in that you created using Corona SDK. Required to pass the exam.

**Submit apps to app stores** – Naming requirements and icon requirements for the various app stores.

This study outline is intended to help experienced Corona developers prepare for the Corona Certified Developers exam. If you would like additional review or preparation materials before taking the CCD, there are additional resources available at <http://www.burtonsmidiagroup.com/books/corona-certified-developer-training-materials/>:

The Corona Certified Developer Study Guide (ISBN: 978-1-937336-10-3) provides a more detailed review of each topic.

The Corona Certified Developer Training Manual (ISBN: 978-1-937336-11-0) provides in-depth instruction into all of the areas covered by the CCD.

Dr. Burton also offers an online course that includes several hours of video instruction on his website: <http://www.burtonsmidiagroup.com/classes/>

