Cool projects that will push your skills to the limit

Corona SDK

A detailed guide with 10 projects specifically designed to expand the fundamentals of this exciting mobile development platform!

HOTSHOT

Nevin Flanagan

[PACKT] PUBLISHING

Corona SDK HOTSHOT

A detailed guide with 10 projects specifically designed to expand the fundamentals of this exciting mobile development platform!

Nevin Flanagan

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Corona SDK HOTSHOT

First published: May 2013

Production Reference: 1140513

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

ISBN 978-1-84969-430-8

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author

Nevin Flanagan

Reviewers

Alan Grace

Sergey Lalov

Michael Piercy

Acquisition Editor

Mary Nadar

Lead Technical Editor

Ankita Shashi

Technical Editors

Ankita Meshram

Veena Pagare

Project Coordinator

Shiksha Chaturvedi

Proofreader

Maria Gould

Indexer

Hemangini Bari

Graphics

Ronak Dhruv

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Nevin Flanagan has had an extremely varied career covering several fields, but the threads of computers, teaching, and games have trailed through it all for years. He has programmed on different levels ranging from assembly language to high-level scripting in game engines, and is credited as a contributor to the World of Warcraft user interface. He is currently fascinated by the interface possibilities offered by mobile touchscreen devices and is completing a Master's degree in Interactive Media and Game Development at Worcester Polytechnic Institute in Massachusetts.

He lives with his wife Jenna in Leominster, Massachusetts, in the U.S.A.

Acknowledgement

Credit goes to Louise and David, my parents, for teaching me both the importance of communicating clearly and the love of storytelling that has driven my passion to play and create games. Eva Farrell, Nadya Murray, and Matthew Lerner joined me in discovering, enjoying, and creating stories; Geoff Gowan, David Larkin, and Mike Weber showed me how to program graphical applications and get excited about making games, and the Lehman Alternative Community School gave us all a place to try it out.

The computer science faculty at Ithaca College gave me tools and opportunities; Chuck Leska showed me the importance of readability; and John Barr provided me with opportunities to use my skills in new fields. The IMGD program at WPI has encouraged me to keep creating new things and pushed me to see how far I can build my ideas; the faculty's confidence and demands have helped me build myself up in the things that matter most, such as vision, commitment, and confidence.

This is a book about Corona, so I must thank Walter Luh and the rest of the team at Corona Labs (including co-creator Carlos Icaza), not only for creating this exceptional tool, but for fulfilling requests and staying engaged with me as I've used their software and produced this book. A huge shout out is also due to the #corona IRC channel on `irc.freenode.net`, especially Lerg, LavaLevel, and lKinx, who gave me people to teach as well as learn with. This book would have been poorer without their questions and advice about their Corona projects.

Finally, immeasurable thanks go to the people in my life who have constantly expressed their support and faith for my ability to contribute something meaningful; especially Lopeppeppy, Unkle, Cairenn, and Onyx, who know who they are; and to Jenna, for lighting up every day of my life whenever I try to smear clouds across the sun.

About the Reviewers

Alan Grace is a co-founder of Pixel Wolf Studios, an Indie game development studio based in Dublin, Ireland. Having worked for a number of years in web and graphic design running his own design studio, Alpha Solutions, Alan has a vast area of expertise across multimedia and game design. Having completed his MSc in Media and Digital Games he set up Pixel Wolf Studios in 2011.

Alan currently lectures on a number of courses teaching game development using Corona SDK. He also was a reviewer on *Corona SDK Mobile Game Development: Beginner's Guide, Michelle M. Fernandez, Packt Publishing*.

Sergey Lalov is a master in radioengineering and programmer from Russia; he got his degree in 2009. Since then he has been working as a network administrator with Linux servers, as a web developer, as a developer of a video surveillance system, and as a developer of an automatic autodrome, where all cars have been equipped with Linux onboard computers with GPS and cameras in a way so people can see in real-time driver's position and score. Finally he became a developer of mobile apps and games for Android and iOS. His brother, Vladimir, is a talented graphics designer and writer. Together they form a great tandem for game development. Now the Spiral Code Studio company has been founded (`http://spiralcodestudio.com`) and they work on a promising futuristic tower defense game—strong science fiction and addictive gameplay that we all love.

Being a game developer has always been a dream job of Sergey's since childhood. As well as many others, he was really impressed when he got his 8-bit NES console (actually it was a Chinese clone called **Dendy**). It was very interesting how this little thing operated and produced dynamic images based on user input. Later at middle school he joined radioengineering club for pupils, where he was first introduced to computers; the club had i286 and i486 machines. His first program was a simple paint-like app for DOS in C. Later there were commodore-like computers with BASIC on board and finally a modern Pentium II computer. At high school he wrote his first simple game for DOS in Pascal—a side-scroller in space, in which the player guided his or her spaceship destroying coming asteroids.

At university he became a web developer and was trying to make a game in 3D using different 3D engines, but only after the university did he find Corona SDK. At that time there were almost no competitors to Corona—it's fast, easy to use, and extremely easy to learn. Having learned Python before, he learned Lua and the basics of Corona SDK in just a week! Lua is a great language, really well thought out. Even now Corona SDK is the most user friendly tool to make fast games for mobile platforms.

Michael Piercy co-founded the Dublin-based, independent game development outfit Pixel Wolf Studios, after achieving an MSc in Digital Games and a BA in Computer Game Design. Focusing on mobile game design and development, he worked on a range of games covering various marketplaces such as iOS and Android platforms.

Michael also worked on the *Corona SDK Mobile Game Development for Beginners Video Series, by Packt Publishing*. His online portfolio is available to the public at `www.MichaelPiercy.ie`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

This book is meant to help you build the skills to create games and other apps for touchscreen mobile devices such as iPhones, Android-based phones, and touchscreen e-readers, as quickly and reliably as possible. We'll aim to avoid spending time on rewriting code, practice structuring your projects to create fewer, more obvious, bugs, and discuss techniques to make your apps friendlier and less frustrating for your end users. In short, we're going to take your experiences with the Corona platform and polish them until you're a Corona hotshot!

We present you with ten original projects in various states of completion to explain and practice the various concepts being presented. Each of these projects was created in two weeks or less using Corona SDK and art assets available under a Creative Commons license from the website `opengameart.org`.

What this book covers

Project 1, *Bat Swat – an Introduction to App Event Cycles*, which is a basic "tap the targets" reflex game, will walk you through the overall lifecycle of a Corona app, from design to polish, and introduce you to the use of custom events to control program flow.

Project 2, *SuperCargo – Using Events to Track Game Progress*, which uses a *Sokoban*-style game to illustrate ways to save game progress and history, allowing the user to undo their moves or resume their game after interruptions.

Project 3, *TranslationBuddy – Fast App Development for any Field*, which shows that not every app made with Corona is a game. This frontend to Microsoft Translate shows how to use Corona's `widget` library to create standard user interface elements, as well as make requests of a remote service.

Project 4, *Deep Black – Processing Mobile Device Input*, is an asteroid-shooting game that introduces you to Corona's physics capabilities, including velocity, force, and collisions. It also presents you with the basics of using the device's accelerometer for user input.

Project 5, *Atmosfall – Managing Game Progress with Coroutines*, helps us learn pans, enemy scripting, and level scheduling with its top-down scrolling shooter. This project also provides useful working examples of class templates in Lua and how they can be used to generate different enemies or weapons from a common core of behavior.

Project 6, *Predation – Creating Powerful Visuals from Simple Effects*, unleashes ZOMBIES! In this project, you'll take a mostly finished game in the style of Missile Command, and learn some techniques for getting extra mileage out of Corona's visual-effects tools to extend it with splattering blood and dissolving monsters.

Project 7, *Caves of Glory – Mastering Maps and Zones*, which is an exploration game, will give us space to explore generating tile-based maps from a custom file format, panning around large maps, and scanning a directory for all available data files.

Project 8, The Beat Goes On – Integrating with Game Networks, is a rhythm game used as a basis for configuring and populating leaderboards and achievements using Corona's game network libraries. Social components in games are the big frontier.

Project 9, Into the Woods – Computer Navigation of Environments, will help you expand your repertoire of tile-based games with path-finding and mobile enemies. You will learn an efficient Lua implementation of the popular A* algorithm and select a path for monsters that follow the player as he or she moves.

Project 10, Underfoot – Selectively Leveraging the Physics System, helps you learn how to control the physics library in all its glory, including selective collisions, detecting solid collisions without bouncing, and controlling gravity on an object-by-object basis.

What you need for this book

Most importantly, you will need a copy of the Corona SDK, as well as a user account for it. The examples in this book were developed and tested against the public release 1076 of Corona, so using them with an earlier version may result in bugs. You can download Corona after signing up for a free account through `http://developer.coronalabs.com/user/register?destination=downloads/coronasdk`, if you don't have it installed already.

We also assume that you've entered into the required agreements with whatever marketplaces you want to publish to, such as Apple or Google. You'll also need to have your agreement in place with Apple if you want to test your apps on iOS devices, and not just simulators. There are numerous Internet resources explaining these agreements, how to set them up, and how much they cost. We assume you know how to create the development profiles and specialized keys that will allow your apps to be distributed.

While they're not required for any of these projects, you're highly encouraged to familiarize yourself with common programming tool applications such as version control and project management tools. An abundance of free software and websites are available, such as git, Mercurial, and gitolite for repository hosting, OpenProj and Redmine for project tracking. These tools can be invaluable for keeping a project on track and moving forward.

Who this book is for

The projects in this book are intended to help people who have grasped the basics of Corona and Lua to move forward, developing more complex projects, using advanced techniques and practicing good software development. We assume that you've already signed up for Corona, and have completed one or two simple projects in it already. The fundamentals of the Lua language are *not* covered, and while the implications or details of a specific API are sometimes discussed, we do not provide complete documentation for Corona's functions or modules.

If you haven't yet gotten started with Corona, you may want to consider starting out with a copy of *Corona SDK Mobile Game Development: Beginner's Guide,* written by Michelle M. Fernandez and available through Packt Publishing.

Conventions

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

Code words in text are shown as follows: "The `timer.performWithDelay` pattern is probably a familiar one."

A block of code is set as follows:

```
self.World:addEventListener('Death', self)
self.World:addEventListener('Despawn', self)

self:addEventListener('Game', self.World)
```

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

```
self.ScoreTotal = 0
self.StartingCount = 1
self.Count = self.StartingCount
```

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

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

Project 1

Bat Swat – An Introduction to App Event Cycles

The biggest challenge with any large project is managing the various elements that make it up and keeping them in sync with each other. Different systems have evolved to help make this easier (object-oriented programming being the most famous). Corona and Lua include features to support object-oriented thinking, but the most prominent feature of Corona that helps the app designer manage communication between program elements is a design pattern, which academics refer to as the publish-subscribe or observer-target model; Corona calls it events and listeners.

What do we build?

We'll start exploring this concept with a simple game, where creatures race across the screen, and you have to tap them before they escape off the other side. While the project design is simple, it provides an excellent arena to test and review the process of translating between low-level events provided by the system, and high-level events that describe events in the abstracted game.

This game project also illustrates how to construct different sections of an app using scenes in Corona's `storyboard` library and how to pass information between them. For instance, the gameplay scene forwards your final score to the menu screen to be considered for high-score status.

What does it do?

The app launches to a menu screen that displays the app name and an instruction to tap the screen. If left unattended, it also displays the high-scores list. Once the screen is tapped, it proceeds to a game screen depicting the ramparts of a tall castle, panning up over the walls as bats fly past. If you can tap the bats before they escape off the screen, they fall off the screen and you gain points. After a certain number, the game reckons up your score based on the number you managed to catch and submits that score to a list of recorded high scores to see if it has earned a place.

How is it excellent?

This system allows multiple objects to be notified when something happens in your program; the important part is that the point where that thing happens doesn't really need to know how many other parts of your code are interested, or what they'll do in response. It also allows a program to be separated into layers which consist of different objects. Each layer defines the events that outside objects might be interested in and adapts between low-level events (like collisions and touches on the screen) and higher-level abstract events (such as enemies dying or creatures coming within range of each other).

You'll be getting a feel for how event-based communications help manage multi-part projects, and you'll do it along with a refresher on how easily Corona lets you assemble a simple, but very playable, game. The logic as presented in the Lua language helps streamline a lot of chores that require many steps in languages like Objective-C.

The project also provides a good foundation for refreshing our acquaintance with some commonly used Corona facilities, notably the `storyboard` library, the `transition` library, and the `sqlite3` database system. All of these will be important as we go forward. This project provides a low-pressure environment to catch up with them and refreshes your memory, which will serve you in good stead.

How are we going to do it?

While the development cycle is very simple, its structure forms the basis for planning basically any project. The structure is as follows:

- Describing the game
- Defining the event flow
- Creating the game scene, and the bat and world objects
- Creating the interface layer
- Adding the shell
- Tracking high scores
- Adding some polish

What do I need to get started?

To complete the core of this project, you will need a small sprite to represent your elusive creatures, in the neighborhood of 32 x 32 pixels. To add the scrolling and parallax backgrounds for gameplay, a large wall graphic (about the size of your mobile device screen) and a distant background image (ideally taller than your target display) are also needed. Additional graphics make the menu screen more interesting.

A complete package of suitable graphics is included in the project download files at `www.packtpub.com`. These graphics, like the others in the book, were obtained from `http://opengameart.org/` and are freely available for reuse under a Creative Commons By license. This allows you to include the graphics, free of charges or royalties, in any project (even commercial ones), as long as the project gives proper credit to the creators of the works used.

Describing the game

The first step to constructing any project, especially any software project, is to document the goals and requirements of the project. A software engineering professor of mine was fond of saying that "if you don't know where you're going, how can you know if you've gotten there or not?" Having a description in writing is a valuable tool for a single developer, because it gives them a way to track all their relevant thoughts and goals and remember what their initial intentions were. It is also an invaluable tool for teams of developers, because it gives them a central source for their task targets and a place to answer many of their questions without taking up time in meetings or in-person discussions for trivial answers.

Getting on with it

Since this project is focused on a couple of simple technical goals, the game design is simple and the description will be short and fairly informal. Create a new text document in an editor of your choice, and enter (or paste) each block of quoted text into it as we review them.

1. A design should start with an overview description that summarizes what the player will experience and why it will be fun:

   ```
   Bat Swat is a game that tests your reflexes, challenging you to
   quickly tap sinister bats as they escape across the screen from
   the bottom left to the top right. Each bat knocked out of the sky
   is worth ten points, and the ten highest scores are displayed on
   the menu screen with the initials of their achievers.
   ```

2. The design progresses to explain the appearance of the game and the gameplay in some more detail:

   ```
   The bats' flight is displayed across the staggered ramparts of
   a gothic castle, which move down and left across the screen to
   show upward progress. A distant landscape is slowly moved down
   in the background to reinforce the impression of upward travel.
   A numerical display of the player's score so far is shown in the
   upper left. A game lasts until 30 bats have appeared on the screen
   and either escaped off the other side or been knocked down.
   ```

3. The design also explains what parts of the game are required in addition to the gameplay screen itself, to meet players' other expectations such as high score lists:

   ```
   Between games, and when the app first launches, a splash screen
   is displayed with the name of the game and a reminder to tap the
   screen in order to begin play. If the game remains on this screen
   for more than a few seconds, it cycles between a display of the
   high score record and the game credits.
   ```

4. The design will explain the specifics of how high scores are selected and displayed:

   ```
   When the menu is displayed after completing a game, if that game's
   final score is within the ten highest scores recorded, a pop-up
   screen is shown displaying the new high score and soliciting the
   player's initials. This disappears and returns to the normal menu
   screen once the initials entry is confirmed.
   ```

What did we do?

Having this body of text handy gives us a touchstone and a definite target. Rather like the adage that you can sculpt by starting with a block a marble and *chipping away everything that doesn't look like an elephant*, our task as a developer is now to correct every point in the project that doesn't conform to our target description.

Committing our thoughts to writing is also important because it requires us to organize our design ideas and clarify ambiguities. Your design document serves as your preliminary rubber duck, a receptive listener to explain your ideas to in order to understand them better yourself.

What else do I need to know?

Many people get intimidated by the thought of writing design documents, concerned about being shackled into their original vision, or being unable to make changes as their ideas develop. It's important to remember that the design document is a component of your project, just like your code and graphics. Just as these components will be developed or replaced, expect your design document to evolve as your project develops. If you are coding and decide something needs to change, you can update your design document to reflect your new intentions.

If you are storing your projects in a version control repository such as git or SVN (which you should be for all but the most trivial projects), it's also an excellent idea to store your design document in that repository so that you can go back and check how the project goals have developed.

Other developers are so excited about their ideas that they view writing a design document as a useless obstacle to seeing those ideas come to life as quickly as possible. While it's important to focus your energies on parts of a project that produce visible rewards and keep you most engaged, the bottom line should be that any code you produce without at least an outline of a design document isn't your project; it's a prototype of your project and you should be prepared to replace any or all of it when your design is finalized. A project written directly from inspiration tends to be composed of pieces that have trouble interacting cleanly, and it will become increasingly disorganized as early testing changes the design.

Defining the event flow

To make sure that the game operates cleanly and the code is maintainable, we will first define the abstract events that take place at game level, and then explain the layer that translates lower-level events into those needed events.

Getting on with it

We need to consider what sorts of events take place at which levels, starting from the top, because the output from the highest-level processes is what ultimately interests us as developers:

1. The game itself has two major events, its beginning and end. It also has one other important event, when the player's score changes; this event is used to keep the score counter in the corner of the screen updated.
2. In order to determine when important game events occur, the game creates, intervenes in, and listens to a world. A **world** is a fairly self-contained system where actions whose results are relevant to the outcome of a game are resolved.

For example, in a game of baseball, the outcome of the game is expressed in scores for the teams, determining a winner and a loser, but these are abstract concepts; the scores are determined by how the rules of the game are applied to things that happen on the field; for example, which hits are caught and which lead to runners on base.

At the most basic level, the outcomes of these actions are determined by details like the laws of physics and players' reach and skill, whether a player can get into position to catch a ball flying in a certain direction, and so on. It is in the context of the world that we know a player has reached a particular base; it is in the context of the game that base is identified as *home plate* and that by reaching it, the player has accrued a run for their team.

3. So, we need to determine what events in our world will be relevant to our game. The game score increases when creatures in the world are defeated by player action. The game is complete once a certain number of creatures have left the world, either by escaping off the borders or by being destroyed. So, the world needs at least two events; one indicating that a creature in the world has *died*, and one indicating that a creature in the world has *despawned*. It's worth noting that every death event will also be followed by a despawn event for the same creature, once the death animation has finished.

   The world generates events for deaths by listening itself to the various creatures spawned into it, waiting for them to post death events to themselves. It also waits for events to be posted for those creatures who are removing themselves from the world, and posts `Despawn` events to itself accordingly.

4. Finally, the creatures themselves are responsible for posting their own death events. Since our game model is very simple, they do this whenever they detect touch events on themselves. Additionally, Corona does not currently post events to objects to inform them when they are removed from the display environment, so the creatures will need to generate those objects themselves when they are ready to leave the world environment.

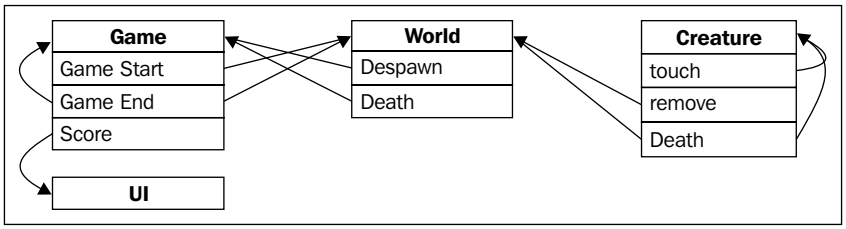
Downloading the example code

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

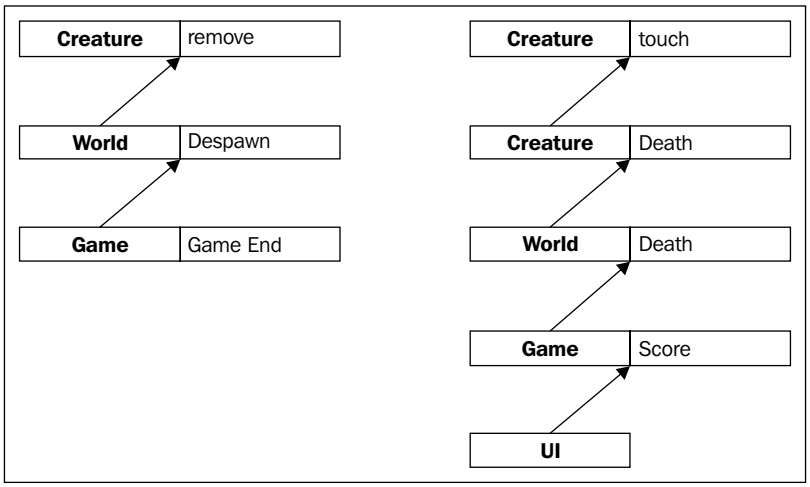
What did we do?

Now we have a verbal description of what layers are required to handle the flow needed to make the game work. In a more formal setting, this would be part of a program design document, which is not the same thing as a game design document; **game design documents** describe game rules, program design documents, and describe how software will be written that implements the game design (or other software; Corona is not just used for games).

Broadly speaking, we now have a plan for events and the layers of the program that they happen at; the plan looks something like the following image (arrows point from events to the objects that listen for them):



Another way to think of the event model for a piece of software is in terms of the event flow; how events occur on specific targets and trigger listeners that dispatch the same or related events to new targets:



What else do I need to know?

This design process, where we start with the most high-level, abstract part of the process and proceed further into the details until we reach whatever the underlying platform gives us, is often referred to as **top-down design**. Designing from the top down helps you build your code's foundation modules based on what the ultimate needs of your app will be, rather than restricting the final program to what you first thought you would need.

The biggest challenge when authoring in a top-down fashion is that you typically create software in such a way that it relies on components that haven't been written yet, so you can't run anything until you fill in the gaps. You can speed up the testing of top-level components by writing what are called **stubs** of the components being used, which are extremely simple functions that provide default responses.

Creating the objects

Now that there's an overall plan, it's time to start making that plan a reality and creating some actual code in a project we can run! As important as planning is, there's no denying that the exciting part of the project is watching the things you've been working on come to life and actually show up on the screen.

Getting ready

Using events in Corona requires tying your conceptual objects to specific types of Corona objects that support events. There are three types of such objects by default: display objects, storyboard scene objects, and the global Runtime object. (It's possible to create your own, but that's not something we're discussing at this point.) The creature objects are visible on screen and need to receive touch events, so it's logical to represent them with display objects. The world needs to hold a bunch of creatures and give them spatial relationships, so using a display group for it makes sense.

The game doesn't have a direct correspondence to anything on the screen, although it's responsible for showing the contents of the world as well as the display of the score. We'll make it a scene, because it has much the same life cycle and because we can use its associated display group to hold both the world and, later, the interface group.

Getting on with it

When you start the Corona Simulator, the splash screen asks whether you want to create a new project, launch the simulator, view your Corona Dashboard for info on apps you've released, or review demos and sample code.

1. Click on the icon for **New Project**; the pop-up screen that you take next changes slightly depending whether you are running the simulator on Windows or Mac OS X:
    - On a Mac, enter the project name and select **Scene** from the template list. Leave the other two entries (size and default orientation) at their defaults and select the **Next** button. Choose a location for your new project file, then select **Show in Finder** from the last pop-up.

- Under Windows, the dialog is somewhat more compact. Enter the project name and select **Multiscreen Application**. Use the **Browse** button near the upper-right corner to select whatever project directory you want to use. The **OK** button will open your project in both Windows Explorer and the Corona Simulator.

2. Once a window is showing the project contents, find the file called `scenetemplate.lua` and make a copy of it. Name the copy `game.lua`.
3. Open the file `main.lua` in the project folder using the text editor of your choice. Find the last line, where it says `storyboard.gotoScene( "scenetemplate" )`, and change it to `storyboard.gotoScene( "game" )`.
4. Save this file. This changes the file which the project will look in for its initial screen content.

Loading art assets and libraries

Download the project pack, if you haven't already:

- Copy the `images` directory and its contents into your project folder. This includes both the sprite sheet for our creatures and the background graphics for our world, as well as some images we'll use later to create our splash screen.
- Also, copy the files `world.lua` and `bat.lua` from the `version 1` subfolder into your directory. We'll include these files in the main game file and discuss their contents later.

Loading the world

Open the file `game.lua`. This file is prepopulated with the skeleton of a storyboard scene, which we'll fill in with code to run the game, and supplement with listener functions to respond to world events.

Find the block for the `scene:createScene` function and start replacing the comment that talks about inserting your own code with several new lines:

```
display.newRect(group, 0, 0, display.contentWidth, display.
contentHeight):setFillColor(0, 0)

  self.World = require "world" {
    Backdrop = "images/exterior-parallaxBG1.png",
    Tile = "images/wall.png";
    Inhabitants = {
      bat = require "bat"
    }
  }
  group:insert(self.World)
```

This loads and calls the `world.lua` module, which generates a new function that constructs a world object with the specified options.

Depending on how much Lua programming you've done, the syntax of the `require` call may or may not be familiar to you; when a name or expression is followed directly by a string literal or table constructor, Lua attempts to call the value of the name or expression (assuming it is a function) with the string or table as its only argument. So the name `require` followed by the literal `world` is equivalent to the function call `require "world"`. Since `require` returns whatever is returned by the module it loads, and `world.lua` returns a function (more on this in a bit), the `require "world"` call itself ends up being equivalent to a function, and followed by the table constructor, the {braces}, and their contents, it becomes a call to that function using that table. The table specifies graphics files to be used by the new world object for its presentation, as well as a list of creatures that the world needs to be able to include and create.

Then, we take the newly created world object and add it to the scene's display group so that it will appear on the screen and be hidden or deleted properly when our game leaves or purges the scene.

Linking the game with the world

Finally, we establish some event trigger relationships between the game object and the world object that it has created:

```
    group:insert(self.World)

    self.World:addEventListener('Death', self)
    self.World:addEventListener('Despawn', self)

    self:addEventListener('Game', self.World)
end
```

As specified previously, the game object will listen to the world for events where a creature has been defeated or has otherwise despawned, so that it can determine when the player scores and when the game should end. It also registers the world object to receive game-related events such as the game beginning and ending. Note that the game has no idea how the world object will respond to these events, or even if it will respond at all. It simply makes sure that the world will be notified when these things happen.

Loading a new game into the display

The `createScene` event is dispatched to scenes only when they are first loaded or if their displays have been unloaded to save memory. When the scene actually begins or is reloaded from a different scene, the `createScene` event might be skipped, but the `enterScene` event always fires.

1. We'll go down to the next event responder, and replace the contents of that function with code that actually starts the game:

```
function scene:enterScene( event )
  self.ScoreTotal = 0
```

2. When a game starts, the player's score should always be reset to 0:

```
  self.ScoreTotal = 0
  self.StartingCount = 1
  self.Count = self.StartingCount
```

3. While the design calls for 30 creatures traversing the screen in turn, first we want to make sure that the basic mechanisms we're creating work. So to start, we will create only one bat so that we don't have to wait for the whole scene to finish in order to try again. While the game only needs to track the remaining count of creatures, we remember the total count we were going to spawn for the game, so it can be stored with high-score information easily:

```
  self.Count = self.StartingCount
  self:dispatchEvent{name = 'Game'; action = 'start', duration =
(self.Count + 1) * 1500}
```

   This is our first custom event that we trigger! This notifies anyone who's listening (like, possibly, the world) that a game is about to begin, and that it's expected to last a certain number of milliseconds. Many games will not have a fixed duration field, but we want to provide an estimate of the game time so that the scrolling parallax background will move continuously throughout the game. This code allows a second and a half for each bat (since that's how often they'll be released), plus another second and a half for the last bat to cross the screen.

Notice that the event's name starts with a capital letter. All Corona built-in events have names beginning with lowercase letters; using uppercase letters to start the names of our own custom events ensures that we won't have any naming conflicts with Corona, and also makes it easy to recognize basic events from our own.

Preparing the game challenges

Once we've triggered the game start, we prepare the bats to be dispatched:

```
  self:dispatchEvent{name = 'Game'; action = 'start', duration =
(self.Count + 1) * 1500}
  for i=1,self.Count do
    local x, y = 0, display.contentHeight,
    timer.performWithDelay(i * 1500,
      function(...)
        self.World:Spawn{kind = 'bat', x = x, y = y, 'brown'}
      end
    )
  end
end
```

This loop prepares the creation of as many bats as the game level is supposed to release. Right now, it will generate only one, because the `StartingCount` event has been reduced to `1` for test purposes; but increasing that number will automatically schedule more bats every second and a half.

The `timer.performWithDelay` pattern is probably a familiar one; it creates a new temporary function on each pass through the loop, which will call another function with the parameters specified in that pass through the loop. The `x` and `y` values specified are currently fixed for testing purposes, but later we will add some variety so that the bats do not all follow the exact same line.

Responding to world changes

Finally, we go up to the top of the scene file after the `storyboard.createScene()` call, and define one more function of the scene object. The scene has been registered as a listener for `Despawn` events on the world, so we need to explain how the game should handle those events when they occur:

```
local scene = storyboard.newScene()

function scene:Despawn(event)
  if not tonumber(self.Count) or self.Count <= 1 then
```

When a table (which includes scene and display objects) is registered as a listener on some event target, it should have a field name matching the event name (including matching case; Lua is case-sensitive), containing a function that will process those events. If it does not contain such a field when the relevant event occurs, it will simply be ignored until the next such event (later, we will use this deliberately). The game is concerned about these events because once all the bats it directed the world to spawn have despawned, the game is complete.

Monitoring game progress

The first thing we do when a creature despawn occurs in the world is check how many more despawns we're expecting. As a sanity check to avoid errors, if the count of remaining creatures is missing or not a number, we also assume that the game is over:

```
if not tonumber(self.Count) or self.Count <= 1 then
  self:dispatchEvent{name = 'Game'; action = 'stop'}
```

Concluding the game

If the count of creatures still waiting to spawn isn't more than one, indicating that there are extra bats still waiting their turn, then the game is over (this count reduces by 1 for each bat that spawns). We dispatch a `Game` event with the `stop` action to notify interested listeners (like the world) that the game is over:

```
    self:dispatchEvent{name = 'Game'; action = 'stop'}
    os.exit( )
```

Once other game elements have performed end-of-game cleanup, we leave the game. Eventually, we will want to return to the menu screen and submit our score to the high-scores table, but that module doesn't exist yet:

```
    os.exit( )
  else
    self.Count = self.Count - 1
  end
end
```

If the game is still waiting for more than one creature to pass, all we need to do is reduce the count we're waiting for by one to account for the count that just passed.

The game module is now ready to test! Before we finish, we can just go down towards the bottom of the file, and remove the skeleton functions for `exitScene` and `destroyScene`, as well as the `scene:addEventListener` calls for them. They will not be needed for this module in this project.

Understanding your libraries

It's worth examining the `world.lua` and `bat.lua` modules briefly, because they implement the rest of the critical event chain that we designed in the second step. Notice in `world.lua` how the world constructor function adds a function to the world object called `self:Game(event)`. Since the game registered the world as a listener for Game events by itself, this function will be called automatically when the game object sends a start or stop event.

The `world` object also provides the `Spawn` function that the game calls to create new bats, and one of the things it does there is add itself as a listener for `Death` and remove events on the newly spawned creature. It responds to `Death` events by reposting them to itself, and to `remove` events by posting `Despawn` events for the game to pick up.

The `bat.lua` module has a couple of points of interest. We've covered in the abstract how custom events can be substituted for physical events; the bat illustrates this directly, by publishing a `Death` event to itself when it is tapped. It then also unregisters itself for future tap events so that you can't repeatedly tap dead bats for extra score.

Notice that the bat object registers for its own `Death` event, as well as the world registering for it. It uses this so that it can separate the cosmetic reaction to its death (currently very simple) from the logical events required to make it happen. It also means that it will respond properly even if something else posts a `Death` event to it from outside.

What did we do?

At this point we have a rudimentary test game that illustrates the flow of events up and down the object layers. For the sake of brevity, we imported existing modules to support two of the three layers we need to deal with, and constructed the third one to control and respond to the middle layer. The game layer does not ever interact with the individual bat objects, honoring the programming principle called **encapsulation** or **weak coupling**.

So far, the game will run a single bat across the screen and then quit. If you manage to tap the bat, it will fade out but finish its trip across the screen. Obviously, this is not a very satisfactory game for the work needed to produce all these files (assuming that you had to write all three of them yourself). However, the next few tasks will show us how comparatively easy it is to incorporate these features into the robust infrastructure we have established.

Creating the interface

While many game designers consider games that require no visible interface to be the platonic ideal of their craft, nearly every game requires some sort of extra-diegetic interface element, something that provides information about the game world and accepts commands into the game world, but is not itself part of the game world. To show the player how well they're doing, we'll add a layer above the game world, with a number showing the player's current score, and increment it whenever their score changes.

Getting on with it

First, we're going to prepare the interface module. This code will live in a separate file to make it easy to maintain without changing the game code, and vice versa, once the initial connection is made. The interface module will be a function that takes a game object and returns the group containing the various interface elements. Start by creating a new text file in the project source directory named `interface.lua` and framing in the outline of the function:

```
return function(game)
  local self = display.newGroup()

  return self
end
```

This gives us the groundwork for a function that returns a new group. Now we can start filling in the body with elements for the group (or one element, in this case).

Adding visible information

First we create the text object that will display the score:

```
  local self = display.newGroup()
  self.ScoreDisplay = display.newText(self, "000", 20, 10, native.
systemFont, 24)
  self.ScoreDisplay:setReferencePoint(display.CenterRightDisplayPoint)
```

Setting the reference point doesn't actually move anything on the screen, but it does change which point in the text is considered by Corona to be the x and y coordinates of the object. This will make it easier to keep the text aligned as we update it later.

Updating an information display

We want the score display to change when things happen in the game, so it will need to listen for the relevant events:

```
  self.ScoreDisplay:setReferencePoint(display.CenterRightDisplayPoint)
  function self.ScoreDisplay:Score(event)
```

Because the score display object will be used as a listener that responds whenever the game's score changes, it will need a function field that responds to `Score` events.

```
  function self.ScoreDisplay:Score(event)
    local x, y = self.x, self.y
```

Because Corona text objects are not naturally aligned, some juggling is needed whenever one might change its text contents and therefore its size. The `x` and `y` values we record here are the ones where we always want the center of the text's right edge to appear.

```
    local x, y = self.x, self.y
    self.text = string.format("%03d", event.total)
```

The format call ensures that the displayed score is always three digits long, with leading zeroes as needed. This keeps the score looking consistent.

```
    self.text = string.format("%03d", event.total)
    self:setReferencePoint(display.CenterRightDisplayPoint)
    self.x, self.y = x, y
  end
```

This ensures that we will be placing the text at its new center-right anchor. Also, it sets that point back to the originally recorded coordinates. Now that we've concluded the function to respond to score changes, we need to register that we're interested in hearing about the scores, before we return the new interface to the game creating it:

```
  end
  game:addEventListener('Score', self.ScoreDisplay)
  return self
end
```

Linking the interface to the game

Save the file and close it. We now have a functional interface layer, but it isn't yet being created or used. Open the `game.lua` file and locate the `scene:createScene` function block. After the lines that create and insert the world group, add similar lines for the interface module:

```
  group:insert(self.World)
  self.Interface = require "interface" (self)
  group:insert(self.Interface)

  self.World:addEventListener('Death', self)
```

We load the interface creator function, and call it, passing it a reference to the game it will listen to for Score events. We then insert the interface into the game scene's display group at a higher layer than the world layer, so that it rides on top (otherwise, the world background would hide the interface).

Triggering a game event from a world event

However, although the interface layer is now being created and displayed, and it's listening for Score events on the game, these events are not being published yet. Near the top of the open `game.lua` file, above the `scene:Despawn` function, add a function, similar in form to the despawn function, to handle death events and modify the game's score accordingly.

```
local scene = storyboard.newScene()

function scene:Death(event)
```

The first thing we will need to do when we detect that a creature has died in the world (rather than just leaving the world borders and despawning) is increment the game score:

```
function scene:Death(event)
  self.ScoreTotal = self.ScoreTotal + 10
```

Then, after changing the actual score, we need to broadcast that the score has changed. This will finish the Death response handler:

```
  self.ScoreTotal = self.ScoreTotal + 10
  self:dispatchEvent{name = 'Score'; total = self.ScoreTotal}
end

function scene:Despawn(event)
```

Lua is a language that's light on semi-colons, making their use between statements optional in almost all cases. In this case, table constructors allow items being added to the new table to be separated with either commas (most of the time) or semi-colons (only occasionally). I like to use them to separate table elements into groups; in this case, I use one to separate the event name (which all events have) from the other parameters (which are particular to each specific event).

Now, the score should increment as you successfully tap the bat flying over (if you want to test it more thoroughly, try changing the game scene's `StartingCount` field from `1` to `5`). The last precaution we need to take is to reset the score properly if the scene is reused for a new game without being unloaded first, by adding an event inside the scene's `enterScene` response:

```
function scene:enterScene( event )
  self.ScoreTotal = 0
  self:dispatchEvent{name = 'Score'; total = self.ScoreTotal}
  self.StartingCount = 5
```

What did we do?

We created a display to show the game score, and set it to update automatically as the game score increases. We made the display aware of the specific game whose score it will display, by passing that game to the interface constructor. We also modified the game to actually produce these events, so that the interface will have some updates to process.

What else do I need to know?

The world object doesn't register itself with the game object for events; it lets the game object do that for it and only provides the response. Why do we instead give the game object to the interface layer and let it register itself?

The main reason is that the world is primarily a source of events for the game to listen to, whereas the interface is primarily interested in events that the game object generates. In a more complex game, it would be very difficult for the game object to anticipate every possible event that the interface might need to display, and every new event the interface wanted to handle would require modifying the game module as well to register it. This approach lets the interface call the shots as far as registration, whereas the game can reasonably assume that it will drive most communication that goes from the game into the world.

Adding the Shell

Most arcade-style games like this one have one or two screens to fill the gap between games, provide instructions, and display score records. Corona's storyboard module makes it easy to implement this splash screen as a separate scene and pass useful cues between the two.

Getting ready

Copy the file `visuals.lua`, from the `version 3` subfolder in the project pack, into your project directory. This is a library that provides some visual effects functions; each one takes the object to perform the effect on, and returns a function that can be called to stop the effect.

Getting on with it

Because most of the code in the menu module is just the straightforward loading and positioning of images, we'll copy the module into the project directory rather than code the entire thing from scratch. Copy the file `menu.lua`, from the `version 3` subfolder in the project pack, into your project directory and open it in a text editor so we can review the points which are more interesting:

```
local scene = storyboard.newScene()

function scene:tap(event)
  storyboard.gotoScene("game")
  return true
end

local function alternate(object)
```

This sets the scene up to be used as a listener for tap events. When the registered target is tapped, the scene will initiate a change to the game scene, which will trigger the initialization of a new game. In order for this to do anything, the scene object must be tagged as a listener (the scene's view group will receive touch events of many of its children that don't handle their own events):

```
function scene:createScene( event )
  local group = self.view

  group:addEventListener('tap', self)

  self.Backdrop = display.newImage(group, "images/splash.png", 0, 0)
```

Creating a staging zone for high scores

We'll add a group on the screen where high scores can be displayed:

```
scene.ScoresWindow = display.newGroup()
scene.view:insert(scene.ScoresWindow)
scene.ScoresWindow.x, scene.ScoresWindow.y = 40, 190
```

This display group, presently left empty, is positioned in the large blank space in the lower-left corner of the splash screen. In later updates it will give us a suitable place to display the high scores list and game credits.

Linking the shell into the play cycle

In order to actually use this scene, we need to adjust the `main.lua` file or it will just continue opening the game scene. Open `main.lua` and change the last line from `storyboard.gotoScene( "game" )` to `storyboard.gotoScene( "menu" )`. This will change the first scene loaded when the game starts.

Finally, now that we have somewhere to return to, we don't have to quit the app when the game is over. You can open `game.lua`, find the line in the `scene:Despawn` function that reads `os.exit( )`, and replace it with the following:

```
        self:dispatchEvent{name = 'Game'; action = 'stop'}
        storyboard.gotoScene("menu")
    else
```

Now the game will loop back to the menu screen after the game is done!

What did we do?

Now we have the basic structure typical of arcade-style games; launch to splash screen, proceed to game, return to splash screen. We not only incorporated a new scene module into the project, but we made the connections to set it as our initial scene and revert back to it after each game is over.

What else do we need to know?

When a display object is touched or tapped on the screen, the object itself is the first one to be informed of the event. However, if it does not mark the event as handled (by returning a truthy Lua value, one which is not nil or false, from the event handler), the event will continue to be passed up to its ancestor groups, in reverse order, then to other objects positioned behind it and their ancestors, until finally it is dispatched to the global object `Runtime` or until one of the objects declares it taken care of. Therefore, a tap on any object which is part of the menu scene will eventually be passed up to the scene's display group, which is the ancestor of all of them.

In scripting languages, **truthy values** are ones that are considered on or positive for `if` statements. In Lua, all values are truthy except for nil and the Boolean value false; even the empty string and the number 0, which are treated as false in many other languages, are considered truthy in Lua.

Tracking high scores

The last ingredient to meet our requirements is high-score tracking. We'll need to pass the final scores from the finished game, collect initials (in true arcade fashion) for new high scores, and maintain the database of saved scores.

Getting ready

Copy the `enterInitials.lua` scene file from the `version 4` subfolder of the project pack into your project directory. This scene is fairly straightforward and adds the pop-up screen that will collect players' initials when they reach a new high score.

This section uses the `sqlite3` library included with Corona to simplify managing score records. While we'll spend some time discussing the intent of the SQL statements included, a detailed discussion of SQL syntax is thoroughly outside the scope of this book. For a good, basic introduction, visit `http://www.w3schools.com/sql/`.

Getting on with it

We're going to start by creating a wrapper module to save the rest of our program from dealing directly with the database. There are three basic tasks that the rest of the program needs the database to do:

- Retrieving a top score
- Determining if a score belongs in the top 10
- Adding a new score to the database

Linking to the database file

Create a new text file in the project directory called `history.lua` and open it. Start by loading the needed library:

```
local sqlite3 = require "sqlite3"
```

This loads Corona support for working with databases and establishes the local name `sqlite3` as your interface for using it. Now it's time to open and prepare the database as needed:

```
local sqlite = require "sqlite3"

local storagePath = system.pathForFile('BatSwat.history', system.
DocumentsDirectory)

local db = sqlite3.open(storagePath)
```

Because the `Resource` directory is read-only on mobile platforms, we store the score database in the `Documents` directory. The `sqlite3.open` call attempts to load the given file location as a new database, and creates a blank database file if no file was found:

```
local db = sqlite3.open(storagePath)
Runtime:addEventListener('system',
  function(event)
        if( event.type == "applicationExit" ) then
            db:close()
        end
  end
)
```

We'll get more into dealing with closed and paused applications in the next chapter, but this just makes sure that the database will be flushed and closed properly if the application is closing.

Initializing the database

We'll need a list in the database where we can store score values:

```
  end
)

db:exec[[
  CREATE TABLE IF NOT EXISTS history (
    happened PRIMARY KEY,
    Score,
    Count,
    Initials
  );
]]
```

This is our first taste of SQL, where we have the database run with the `:exec` method. It just makes sure that the database has a table to hold our high scores, including when the score was achieved, how much it was, how many creatures there were on the whole level, and who got the score in question. The time the score that was achieved is used as the table's **primary key**, meaning there can be only one score for any given moment of completion. As the game is single-player, this is fine. If the database already existed, this line will do nothing.

Notice the use here of the Lua long string literal, enclosed in double brackets. This form ignores escape characters, quotes, new lines, and everything else except its closing bracket, simply treating them as characters in the string. This makes it ideal for incorporating code from other languages, reducing the likelihood that some element from the stored code will end the string prematurely or be misinterpreted by the Lua parser.

Cleaning up old scores

We don't need to keep more scores than we can display.

```
  );
]]

db:exec[[
  DELETE FROM history WHERE happened NOT IN (SELECT TOP 10 happened
FROM history ORDER BY Score DESC)
]]
```

This is a maintenance line. We don't want the list of high scores to just keep getting bigger and bigger and taking up more of the user's device memory, so whenever we launch the game, we clean out any scores that have fallen off the bottom of the list. The SQL statement here basically says, "make a list of all the times that have scores in the top 10, and then delete every score whose time isn't on that list."

```
]]

local history = {}

return history
```

Here, we're just preparing the `history` module that will be returned when the file is loaded. The three functions that are the substance of the module will be added between these two new lines, since the return must go last.

While programming styles vary, it's frequently a good habit to build your code inwardly; for instance, when you type the beginning of an `if … then` statement in Lua, you can immediately type `end` on the next line and then back up and fill the contents in between the two. This approach helps you avoid forgetting to close function calls, long strings, loops, and other things that can end up unbalanced.

Considering possible new high scores

To see if a new score qualifies as a high score, we'll see where it falls among the scores already gathered:

```
local history = {}

function history:find(score)
  for count in db:urows([[ SELECT count( ) FROM history WHERE Score >=
]] .. score) do
    return count + 1
  end
end
```

This function identifies where a proposed score would fit into the list. We'll use it to identify which new scores have a place in the top 10 and should ask for the player's initials. It works by counting the number of existing score entries that are larger than the score under consideration.

The `sqlite3` library offers three different functions for scanning through the results of a `SELECT` query. The `db:nrows` method returns a table representing the row, with named fields matching the column names that hold their values for that record. The `db:rows` function gives back a table which simply holds the value of the first column at index 1, the value of the second column at index 2, and so forth. The `db:urows` function, used here, simply uses Lua's multiple returns to pass back all the values from the record without making a new table, in the same order they would appear in the table returned by `db:rows`.

It's worth noting that all three functions don't actually return any records; they return iterator functions that produce the contents of a new row each time they're called. This makes them ideal for use in Lua generalized `for` loops.

Saving new high scores

When a new score is identified as being better than a previous high score, we need to record it:

```
end

function history:add(statistics)
  if statistics.HighScore <= 0 then return end
```

This function will submit a new score to the database. If for some reason a score of `0` was submitted, we won't bother storing it:

```
if statistics.HighScore <= 0 then return end
db:exec(
  string.format([[
    INSERT INTO history VALUES (
      datetime('now', 'localtime'),
      %d, %d, %q
    );  ]],
    statistics.HighScore,statistics.MobCount, statistics.Initials
  )
)
```

Here, we use the `string.format` function (a close relative and derivative of the C `printf`) to fill in the specific information provided to us about the score into an otherwise preprogrammed `SQL INSERT` command. Executing the finished command adds the new row into the database.

Recovering old high scores

To display the scores, we'll need to retrieve them from the database:

```
  )
end


function history:TopScore(index)
```

The last function we add will retrieve the score with a given index; 1 for the highest score, 2 for the second highest, and so on:

```
function history:TopScore(index)
  local query = [[SELECT * FROM history ORDER BY Score DESC LIMIT 1
OFFSET ]] .. (index - 1)
```

Here we prepare the query. The `LIMIT 1` clause means we only want one value from the list, and the `OFFSET` clause indicates how far down the sorted list we want to find that value, basically like an index into an array:

```
  local query = [[SELECT * FROM history ORDER BY Score DESC LIMIT 1
OFFSET ]] .. (index - 1)
  for info in db:nrows(query) do
```

To make it easier for the code calling this function to use the result, we use the `db:nrows` iterator to get back a table structured like a record, with named fields for the column values.

```
for info in db:nrows(query) do
  return info
end
```

Like most database functions, the `luasqlite3` iterators aren't really intended to be used with single values. We could save the obtained record in a variable local to the function and trust the loop to exit after one pass (since the query statement specified should never return more than one record), but just returning out of the loop on the first pass also works fine, since we have nothing else to do after finding the first record.

Since the function will always return from the first pass through the loop, there's no need for any other body, and we're done with the module:

```
    return info
  end
end

return history
```

Communicating scores between modules

Now that the score tracker is ready, we need to prepare the other modules to use it. First, we make a small change to the game scene file, to make it pass its final score back to the menu scene for consideration. The `storyboard` library has added the ability to hand parameters off to a scene when you load it which is perfect for this purpose:

```
function scene:Despawn(event)
  if not tonumber(self.Count) or self.Count <= 1 then
    self:dispatchEvent{name = 'Game'; action = 'stop'}
    storyboard.gotoScene("menu", {params = {Score = self.ScoreTotal,
Count = self.StartingCount}})
  else
    self.Count = self.Count - 1
  end
end
```

This way, the menu's `enterScene` function will be able to access the score and count as fields of the `event.params` table. This is the only change we need to make to the `game.lua` scene file. Next, open the `menu.lua` file to add support for receiving this data. Start by loading the history module at the top of the file so that we will be able to check whether the received score is a new record:

```
local history = require "history"

local storyboard = require( "storyboard" )
```

Now, we need to add support for using that module to check for a new high score and record the initials that qualify. If we have to pop up a collection window, we want to hold off on running any animations until we've returned from that process, so replace the unconditional call to the `scene:Cycle()` function, which handles showing the high scores and credits, with a conditional statement:

```
function scene:enterScene( event )
  self.Banner.alpha = 0

  if event.params and event.params.Score then
    if history:find(event.params.Score) <= 10 then
      storyboard.showOverlay("enterInitials", {effect = "fromBottom",
params = {Score = event.params.Score, Count = event.params.Count},
isModal = true})
    end
  else
    self:Cycle()
  end

end
```

Reviewing new scores

First, we check whether we've received a score at all. Remember that this scene is also launched when the app starts up, in which case there will be no new score to forward.

Next, it uses the `history` module to ask whether the newly received score belongs in the top 10. If not, it won't be showing the initials entry screen and can go directly to running animations.

If this is a new high score, however, we need to display the `enterInitials` pop-up scene to collect user input. We use the `storyboard` library's `showOverlay` function to display the new scene over the current one, since we will be coming straight back to the splash screen when we are done. We pass scene and count to this function, just as we received them, so that the data entry screen can record them in the database. The `isModal` argument field prevents touches in the pop-up scene from drifting down into the menu screen while it is active.

Finally, we register the menu scene to notice when the score is recorded and the overlay is closed, so that it can start its animations. First, we specify that the `Cycle` function (which runs those animations) should be the scene's response to any overlays ending; then we make sure the scene knows that it is interested in its own `overlayEnded` events:

```
end

scene.overlayEnded = scene.Cycle
scene:addEventListener( "overlayEnded", scene)

function scene:exitScene( event )
```

Displaying the score history

Now that score processing is ready, we're going to add code to actually display the high scores. For the moment, we'll just lay them out in the designated space as soon as animations are visible. So, we'll add that call to the menu's `scene:Cycle` function:

```
function scene:Cycle()
  self.StopPulsing = visuals.PulseInOut(self.Banner)
  self.StopEffects = revealScores(self)
end
```

Because we're expecting this to be animated later, we're leaving open the option to have a transition that we might need to stop or change. Right now, we'll focus on just making the scores show up in the new `revealScores` function:

```
local function revealScores(scene)
  display.remove(scene.ScoresSlide)
  scene.ScoresSlide = display.newGroup()
  scene.ScoresWindow:insert(scene.ScoresSlide)
end

function scene:Cycle()
```

This adds a new group to store all our high score displays in, making it easy to animate or clear all of them at once. Before that, however, we remove any previous high-score displays to make room, since the high scores may have changed since they were last displayed:

```
  scene.ScoresWindow:insert(scene.ScoresSlide)
  for i = 1, 10 do
    local score = history:TopScore(i)
```

Next, we loop through the 10 highest scores in the history of the game. The score variable will actually be a table containing all the relevant fields:

```
    local score = history:TopScore(i)
    if not (score and score.Initials and score.Score) then break; end
```

If the game is new, the high score table might be mostly empty, so if we run out of scores, we finish the loop early:

```
    if not (score and score.Initials and score.Score) then break; end
    display.newText(scene.ScoresSlide, score.Initials, 0, 24 * (i -
1), native.systemFont, 16)
    local score = display.newText(scene.ScoresSlide, score.Score, 0,
24 * (i - 1), native.systemFont, 16)
```

We create the two text objects to hold the initials and the actual score. Creating two objects means that they can be aligned separately:

```
    local score = display.newText(scene.ScoresSlide, score.Score, 0,
24 * (i - 1), native.systemFont, 16)
    score:setReferencePoint(display.CenterRightReferencePoint)
    score.x = 100
  end
end
```

Finally, we align the score number on the right-hand side of the available space. The score reveal is now basically complete!

What did we do?

By recording high scores and allowing people to compete for the best, we've finished adding the core criteria required by the design document. Some features that were described aren't implemented yet, but they're all fairly cosmetic in nature. That means the game is functionally finished, and now is a good time to test it out. Play it repeatedly and look for anything that seems broken. Or, just keep playing it for a while; you've earned it!

Adding finishing touches

Although the game is functionally working at this point, it's not really ready for the prime time. The way the bats just fade out and keep flying could easily confuse players. The high scores table could use a little more excitement. Finally, since we're reusing Creative Commons art assets, some credit information is in order.

Getting ready

Copy the `explosion.lua` file from the `version 5` subfolder of the project pack into your project directory. This file provides the sprite sheet info and a simple function to create and animate a small explosion at a given point.

Getting on with it

First, we're going to make the bat's death a little more dramatic, adding a little explosion and causing the bat to fall off the bottom of the screen instead of continuing to fly. Open up the `bat.lua` file and find the `mobDied` local function, which is registered to go off in response to the creature's `Death` event, and delete the line that says `self.alpha = 0.3`:

```
local function mobDied(self, event)
  transition.cancel(self.Flight)
```

The first thing we're going to do is stop the bat's normal flight course:

```
transition.cancel(self.Flight)
explosion(self.parent, self.x, self.y)
```

We'll use the new explosion module to create an animated explosion in the bat's world environment at its current point. The explosion is responsible for animating itself and deleting itself when the animation is done.

Changing the creatures' motion

We want the bat to seem to fall when it is killed:

```
explosion(self.parent, self.x, self.y)
transition.to(self, {time = 1000, x = display.contentWidth})
```

We'll start the bat sliding sideways off the right-hand side. Because we're looking for a natural bouncing motion under the effects of gravity, the horizontal aspect of the motion will be tweened linearly, while the vertical component will be tweened quadratically; this means we need two separate transitions:

```
transition.to(self, {time = 1000, x = display.contentWidth})
local distance = 100 + (display.contentHeight - self.y)
```

We calculate the total distance the bat will travel vertically to rise up 50 pixels and then fall off the bottom of the screen. We add 100 because it will also fall the extra 50 pixels that it bounced up:

```
local distance = 100 + (display.contentHeight - self.y)
transition.to(self,
  {
    time = 1005, transition = arc, onComplete = clean;
    y = display.contentHeight
  }
)
```

Animating on a custom curve

Because the bat will first bounce up a little, then fall, its vertical motion will be determined by a custom tweening function, `arc`, which we will write in a moment. This transition is made slightly longer than the first one, because it will clean up the object (and post a remove event) when it's done, and we want to make sure the other transition has finished first:

```
local function arc(t, tMax, start, delta)
  if t <= 250 then
    return easing.outQuad(t, 250, start, -50)
```

```
    else
      return easing.inQuad(t - 250, tMax - 250, start - 50, delta + 50)
    end
  end

  local function mobDied(self, event)
```

This is our custom tweening function. It isn't terribly fancy; it transitions the object back, against the direction of the tween, for the first 250 milliseconds, then tweens it forward from that point for the rest. It relies on Corona's existing quadratic tweens (a decelerating tween for the upward motion, and an accelerating one for the fall) to calculate the intervening values.

Adding visual interest to the high scores

Now the bat dies a little more dramatically and we can move on to animating the credits and high scores. Close `bat.lua` and open `menu.lua`, and add a new text object to the `scene:createScene` function:

```
    self.ScoresWindow.x, self.ScoresWindow.y = 40, 190

    local creditsText = [[
Bat Swat code:
  Nevin Flanagan
Images:
  Bat Sprite:
    MoikMellah
  Environment art:
    Jetrel
  Licensed from
  opengameart.org
  under CC-By 2.0]]
  self.Credits = display.newText(self.ScoresWindow, creditsText, -16,
0, 190, 144, native.systemFont, 12)
end
```

This credits object will trade places periodically with the high-scores display, so whenever it fades out, it needs to cue the high-score object to slide into the empty space:

```
  self.Credits = display.newText(self.ScoresWindow, creditsText, -16,
0, 190, 144, native.systemFont, 12)
  self.Credits:addEventListener('Faded', function() self.StopEffects =
visuals.SlideInFadeOut(self.ScoresSlide) end)
end
```

We don't want the credits to appear until the high scores have faded out, so make them completely transparent whenever the scene starts:

```
function scene:enterScene( event )
  self.Banner.alpha = 0
  self.Credits.alpha = 0

  if event.params and event.params.Score then
```

This means that in order for it to ever appear, it needs to be cued in whenever the high scores fade out, which we'll set up in the `revealScores` function:

```
    score.x = 100
  end
  scene.ScoresSlide:addEventListener('Faded', function (...) scene.
StopEffects = visuals.SlideInFadeOut(scene.Credits) end)
end
```

In order for the object to ever receive a `Faded` event, we'll have to start the same transition on it:

```
  scene.ScoresSlide:addEventListener('Faded', function (...) scene.
StopEffects = visuals.SlideInFadeOut(scene.Credits) end)
  return visuals.SlideInFadeOut(scene.ScoresSlide)
end
```

This moves the collected scores down off the screen, and schedules them to slide back in after a second and a half. Returning the resulting cancellation function means that the `Cycle` function will store it in the `scene.StopEffects` field. This is important, because if we start a game, we need to be able to cancel that transition. We can do that from the `scene:exitScene` function.

```
function scene:exitScene( event )
  self.StopPulsing()
  if self.StopEffects then
    self.StopEffects()
  end
end
```

Parameterizing the game length

Finally, we need to update the number of bats per scene; right now, it is set to a test value of 5. The document specifies 30 per game. To leave ourselves room for flexibility in the future, we'll have the menu pass in the desired number when it calls the game scene, much the same way the game passes the final score out. Go to the top of `menu.lua` and add the following line:

```
local scene = storyboard.newScene()

local options = {params = {Count = 30}}

function scene:tap(event)
```

We'll reuse this table as we relaunch the game scene. Now, we just need to supply it when we launch the game, which is in the function right under that:

```
local options = {params = {Count = 30}}

function scene:tap(event)
  storyboard.gotoScene("game", options)
  return true
```

Adding a reference to the options table tells `storyboard` to pass the parameters in to the scene being opened. Finally, we can save and close `menu.lua`, open `game.lua`, and make one change to the `scene:enterScene` function:

```
function scene:enterScene( event )
  self.ScoreTotal = 0
  self:dispatchEvent{name = 'Score'; total = self.ScoreTotal}
  self.StartingCount = event.params.Count
  self.Count = self.StartingCount
```

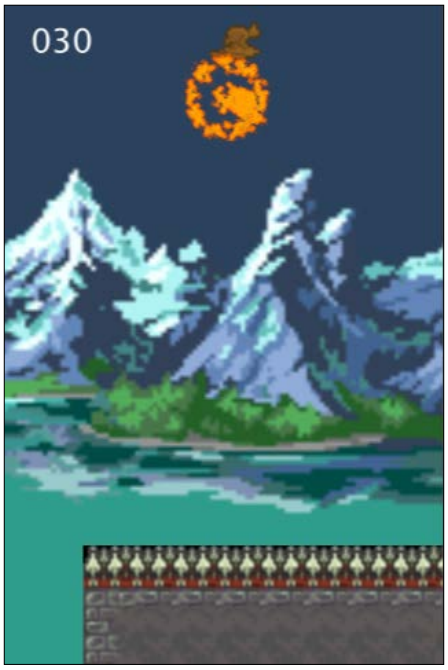
Instead of setting the game's starting creature count to a fixed 5, we'll set it to whatever is passed in by the menu.

What did we do?

Now, we added additional animations to the dying bats by modifying only one function (and adding another one for it to call). We also added a little animation to create visual interest and made control of the game scene more flexible.

Game over – wrapping it up

We have engineered a game from concept description up through final polish and improvements. We have designed and implemented a flexible system for bridging communications between disparate elements of the program and adapted the components within that system to accommodate new elements as they are included. We also now have a playable game! How many bats can you swat?



Can you take the HEAT? The Hotshot Challenge

There are several generalizations that have been left in the code to make it easier to add new components and variations later with a minimum of fuss. Can you give the player a choice between the existing level and one that mixes the brown bats with black ones that move slower, but take three taps to kill? For the cleanest results, try and do it without creating any new scene files!

Project 2

SuperCargo – Using Events to Track Game Progress

Mobile games and apps bring a new challenge to developers; they are frequently used in open spaces, on the go, while waiting for something, or in other circumstances where the user might be interrupted at any time. Some of the most common examples are as follows:

- The user's meal (train, or person they're meeting) might arrive, causing them to put their device away
- The user might be playing on a device with telephone functionality, and receive a call in the middle of the game
- The user might switch to another app, causing your program to be closed to free up device memory
- The user's device might run out of battery power

To be popular with users, apps have to "preserve state", that is, remember where they are at each given moment, so that if one of these or any other interruption occurs, they are not frustrated by losing their progress. We'll explore how the event model that we discussed in *Project 1*, *Bat Swat – An Introduction to App Event Cycles*, can help us streamline the process of remembering what the user has done so far.

What do we build?

To illustrate these principles, we'll make a clone of the popular puzzle game *Sokoban*. This game originated in Japan, where the title means *warehouse keeper*, and was popular among Unix users. Ports exist for many systems, including several mobile versions. The goal is to arrange boxes into target spaces through a confined space, without pushing any box into a corner that you can't get it out of. We'll cover the details in the game design document.

What does it do?

Like *Bat Swat*, *SuperCargo* will follow the basic splash screen-game-splash screen cycle of most arcade games. By the time we're done, the splash screen will allow the user to select one of several levels to play through. The game content will load the chosen level and display it using a tiled sprite sheet, then receive game moves from taps on different parts of the screen. The game engine will determine whether these moves are illegal, process the results of legal moves, and display them. The game will also track the number of moves used to progress through a given level and allow users to undo moves, so that they can escape from unwinnable conditions or consider alternate strategies. Finally, the game will remember progress through uncompleted levels, including more than one level at a time, to give the user more flexibility.

Why is it great?

Sokoban has a well-developed set of file formats and a huge body of existing levels, so this game would be easy to expand with in-app purchases or by allowing the user to add their own level files from public sources, so a lot of potential value is possible at no cost. For us as developers, however, the real benefit comes from seeing how easy it can be to preserve game progress by storing user input streams or other game events. The event model that we developed for *Bat Swat* means that we will actually be able to add saved progress with only the smallest of code changes.

How are we going to do it?

A couple of the tasks from the first project will be condensed to give more focus to the others, and because they consist of more familiar tasks such as:

- Describing the game
- Loading a level from a file
- Displaying the map contents
- Adding the interface
- Making the game playable

- Adding the shell
- Supporting `undo`
- Preserving game history

What do I need to get started?

You'll need two main resources, a levels file that contains text descriptions of the various levels, and a tiled image file that contains icons of the walls, empty floors, goal spaces, boxes, and the player. To ensure proper scaling, each tile in the image file should have a one-pixel border around it on all sides that repeats the pixels at the edges of the usable area. Without this, when using Corona's auto scaling to support higher-resolution displays, you may notice seam lines or a grid where the tiles don't quite touch each other.

The included image file, `bomb_party_v4.full.png`, was generated from a downloaded sprite sheet by using ImageMagick's `convert` tool, a command-line tool highly useful for carrying out automated processing of images, including cropping, scaling, and layering images together. While a discussion of ImageMagick command syntax is well beyond the scope of this book, you can read many useful tips at `http://www.imagemagick.org/Usage/` or review the `tileextrude.sh` file we've included in the code bundle for this project.

Describing the game

This step can't be overlooked! Even though it's not part of the coding and might feel familiar from last time, it's critical to start every project with a clear description of what that project needs to accomplish.

Getting on with it

Let's go over the various sections of the `design.txt` file included with the project pack and consider what each one tells us about the project.

Core mechanic

We'll start by laying down the basic principle behind the game's creation, and a summary of its rules.

Supercargo is a mobile port of a popular puzzle game, wherein the player controls an onscreen avatar that can move in the four cardinal directions. The avatar's progress is blocked by walls that completely bound the area and create shapes inside that boundary. There are boxes distributed throughout the area, and if the player attempts to move their avatar onto one box, it will push the box if there is an empty space across the box from the player's avatar. Boxes cannot be pulled or slid across the avatar's direction of motion. There are a number of goal spaces throughout the game area, equal to the number of boxes; when each box is placed on a goal area and each goal area is occupied by a box, the level is complete. Players' completion of a level is scored according to the number of distinct steps taken by their character to achieve a solution, with lower scores being more highly rated.

This is the fundamental design summary. It summarizes the core rules of *Sokoban*: there is one player who can move in the four major directions; the player can push boxes if there is room, but not pull them; there are goal spaces to take the boxes to, with one for each box, although any box can go on any goal. It tells us that there are two major kinds of features that don't move (walls and goals) and two kinds of features that do move (the player and boxes). We'll use this strategy when designing the game internals, to separate the map into a moving layer and a non-moving layer.

This section also explains the win condition; for each box in the "moving" layer, there is a goal at the same point in the non-moving layer, and vice versa. We'll need this in the fifth task, *Making the game playable*, to recognize wins.

This also says that we'll need to keep track of the number of moves, to determine how good the solution was. We're only going to count moves that change the board (move a box) and not every step the player takes moving into position.

Interface summary

Let's go into more detail about how the user will use the device to control the game.

To receive player input on mobile platforms, input of desired directions of movement will be accomplished by tapping near that edge of the screen. In exploring possible solutions, undo is a critical feature, so undoing the last move that pushed a box will be possible by shaking the device. A counter in the corner of the screen will indicate how many moves the player has taken to reach the displayed game state. If the screen has not been touched for several seconds, the display fades until the next touch to allow for unimpeded contemplation.

This describes the various mechanisms that the user will use to interact with the game. It tells us that we need to be able to recognize taps on different areas of the screen, but not on specific objects, which will be important for the interface layer. It says that we have to recognize shakes of the device, and that we have to be able to back up to the condition before certain steps.

It also describes the informative features of the interface (very minimal in this game) and a feature to make the game feel more responsive. However, this feature will be entirely contained in the interface; the core game does not need to know whether the move count is visible to the user, it only needs to know what the move count is.

Persistence requirement

Next, we specify the requirement we discussed at the beginning of the project, to save progress.

To facilitate preserving progress across multiple sessions, the game will save a move list and update it for each move added. When the user opens a level for which a move history is in progress, the program will ask them whether they wish to resume or discard that game. Completing a level discards its accumulated history.

Some of this was implied by the need to undo, but we'll need a *persistent* move history, one that we can reload if the game is run again later. This also explains what will happen if a level was not completed when the game was interrupted. This is the core design goal of this project, but because the rest of the game needs to be completed first, it will wait for the last task.

Data format

The format is well established, but let's confirm that we subscribe to the same definition.

Levels will be loaded from text files in a format standard for this type of puzzle game. Each line of a text file represents one row of the grid on which the game is played, where a `#` *character indicates a square filled with a wall, a* `.` *(period) character indicates a goal space for a box, a* `$` *indicates a box not on a goal space, a* `*` *character indicates a box initially positioned on a goal space, a space character denotes blank floor, an* `@` *character indicates the player's starting avatar position on plain floor, and a* `+` *character indicates the avatar's starting position on a goal space. A level can contain exactly one* `@` *or* `+` *character, and the number of* `$` *characters should equal the total number of* `.` *and* `+` *characters.*

This is a common text format used to store *Sokoban* levels, referred to as a `.sok` file. For this reason, the module that will translate portions of the files into in-memory representations of playable levels will be called `sok.lua`. This guideline also tells us that there can be single spaces in the level description (`*` and `+`) that affect both the static portion of the level (goal spaces) and the movable part (player or boxes). This will inform the method we'll use to process the file in task two, *Loading a level from a file*.

Additional data requirements

A single text file can contain multiple levels. A level consists of all consecutive lines that contain only characters recognized as part of the level format (`[#$.+*@ ]`). The convention is that each level will be preceded by a line containing its sequence number in the file, and followed by a tilde (~).

This also tells us about the level data format. We'll need to know which level is desired when loading from a file. We'll also need to be prepared to parse through multiple levels and keep track of which one we're processing.

Preliminary module design

Now that we understand the written design and some of its ramifications, we'll outline what components are needed and what events they'll use to communicate.

The major components will be `Game` and `Shell`. These represent two distinct modes, and the app will only be in one at any given time. For this reason, and because they each have distinct displays and modes of interaction, they will be represented by two different storyboard scenes in Corona as follows:

- The `Game` component will need the following components:
  - The `Map` component will store the internal representation of the game state: walls, goals, player, and box positions
  - The `World` component will display the state of the `Map` component to the player
  - The `Interface` component will show non-world information from the player and collect player input for the `Game` component
- The `Shell` component will need components for the following tasks:
  - Selecting a level
  - Starting a game with the selected level
  - Identifying whether to resume or restart an incomplete game

What did we do?

We started by explicitly describing the particulars of the design. We made notes of different requirements of the design, and guesses about what we need to do in development to support these requirements. Finally, we summarized how we can break down our tasks in order to effectively implement each one.

Loading a level from a file

During the design process, we identified a *Sokoban* level as a two-dimensional array with two layers, one of moving elements and one of static elements, which can interact with each other (for instance, although the player and the boxes can both move over goals or empty spaces, neither can move into wall spaces). We have a choice of whether to define the array as **row-major**, where each entry in the parent array is a row, represented as an array of column values; or **column-major**, where each entry is an array that stores all the rows for a given column.

Lua doesn't directly support two-dimensional data structures the way some languages do. But it does support making arrays (tables with sequential integer keys) where the value of each slot is another array.

Since the level files represent each row of the map with a line of text, they are naturally row-major in their storage; therefore, we will use a row-major format for our world arrays. There will be two such arrays, one for moving objects and one for static elements; they will be stored as fields in another table that represents the map. In keeping with this, we'll refer to the indices as row and column as much as possible, to minimize confusing them with the usual order referring to x and y coordinates.

Every space on the map is either an empty space, a wall space, or a goal space. So each value of a column entry in row arrays belonging to the static layer will be one of the strings `#`, `.`, or (space).

Some spaces contain boxes, one space contains the player, and some spaces contain nothing that moves. So each value of a column entry in row arrays that belong to the moving layer will be one of the string `'@'`, the string `'$'`, or the value `nil`.

Getting ready

If you haven't already done so, create a new scene-based project using the Corona splash screen, the way you did in the first project. Then fill that project in with a few pre-made pieces:

1. Copy the `images` and `levels` folders and the `interface.lua` and `game.lua` files from the `version 1` folder in the project pack of this new project.
2. To load the game scene when the app starts, open `main.lua` and change the `storyboard.gotoScene( "scenetemplate" )` call to `storyboard.gotoScene( "game", {params = {File = "levels/default.txt", Level = 1}} )`.
3. Finally, open `game.lua` in a text editor.

Getting on with it

Before we write the `sok.lua` module, let's take a look at the module that uses it, `game.lua`. While the beginning of this file contains mostly basic graphics and sprite definitions, notice that it starts by requiring the `sok` module. The meat of this module is, like most scenes, in two functions: the `createScene` method and the `enterScene` method. It also continues in a familiar fashion: it loads a `World` module to display the state of the game model and registers `World` to receive the `Game` events on itself; and it loads an `interface` module to handle user input, registering itself to receive the `Move` events from `interface`.

In the *Bat Swat* project, user input was received and processed by the world, and all the interface did was display the score. Why is the interface processing input in this project?

In *Bat Swat*, the user was interacting directly with specific objects in the world, so it was much easier to receive the input directly at those objects. In the Supercargo design, the user's touches are on zones of the screen; the user isn't interacting with the specific things in the part of the screen he's/she's tapping on, but just indicating direction.

The `enterScene` method contains the call to load the game map, using the `sok` module required earlier. The `Map` module is an abstract structure, not tied to any particular on-screen object, and used by the game to track the state and determine legal moves.

This pattern, where a purely abstract `Map` is used by logic in a semi-abstract `game` object to adjudicate input from a concrete, on-screen `interface` object, is an example of what is called **model-view-controller**, or **MVC** programming, where the view is responsible only for displaying states, the model is responsible only for tracking states, and the controller is responsible for managing all the interactions and messages that pass between the two. While it's a fundamental principle in programming for Apple's platforms in Objective-C, MVC thinking is useful in any language, because it helps separate tasks that are only loosely related and track all possible points of communication.

After loading the `map` object, the game tells the `world` module to display its contents using the `world:Load()` method and sends an event that the game is now ready to begin. It includes a reference to itself so that listening modules such as `world` can register for any other events that they might need to be informed of, without the game needing to know what those events might be.

Writing the loader

But let's take a closer look now at the `sok.load()` call, which dictates what the modules we're about to write will need to provide. The calling module provides the path to the file containing the levels (a string), as well as the index of the desired level within the file (an integer). In return, the game expects back `Map`, a table object formatted as we described at the beginning of the task. So, our module needs to include a function named `Load`, which accepts those two arguments (a path and an index) and returns the constructed `Map`. The following are the steps to write the loader:

1. Create a new file in the project directory, `sok.lua`, and start by filling it with the basic module skeleton:

```
local sok = {}
return sok
```

2. A Lua module should return a usable object. This is most often a table with functions or configuration values as fields, but it might also be a function that carries out some self-contained task. So, we start by creating and returning the table. But we've also established that the module needs to contain a `load` function with two arguments:

```
local sok = {}
function sok.load(fileName, index)
end
return sok
```

3. We know that the function needs to create and return a table with two layers, so let's establish that:

```
function sok.load(fileName, index)
  local self = {
    Fixed = {};
    Moving = {};
  }
  return self
end
```

Parsing levels from the file

A level is defined as any block of consecutive lines in the file that all contain level content, where the lines before and after, if any, contain non-level content. The following are the steps to parse levels from the file:

1. As we read through the file, we'll need to track which level we're processing as well as whether we're crossing the line from one kind of line to the other:

```
    Moving = {};
  }
```

```
  local discoveredLevel, lastLineIsContent = 0, false
  return self
end
```

2. Then, we use the Lua interface's `io` library to look at each line of the file one after another:

```
  local discoveredLevel, lastLineIsContent = 0, false
  for line in io.lines(fileName) do
  end
  return self
```

Check your line endings

When you're processing text data files, make sure you know the line ending format used in the file! Lua, by default, processes files assuming they use Unix/Linux/OS X line endings (which are also the standard for iOS and Android), but if you develop using Windows, your text editor may use a different format by default!

3. For each of those lines, check whether it contains only content characters:

```
  for line in io.lines(fileName) do
    if line:match("^[# .@+$*]+$") then
    end
  end
```

This is our first example of using Lua *patterns* to recognize whether a string meets certain rules. Patterns are like *regular expressions* from languages like perl, except that they give up a few features in exchange for being much higher-performance. The preceding pattern says:

Does the entire line consist only of the characters `#`, `@`, `+`, `$`, `*`, *(space), and* `.` *(period)?*

4. If we have a content line, then we need to see if it follows another content line (in which case it is part of the level already being considered), or if it is the first line of a new level. Either way, we need to remember when processing the next line that this one had level content.

```
    if line:match("^[# .@+$*]+$") then
      if not lastLineIsContent then
        discoveredLevel = discoveredLevel + 1
      end
      lastLineIsContent = true
    end
```

Parsing the desired level

As we loop through the file increasing the discovered index, eventually we should reach the level that was requested, and process it if so. The following are the steps to parse the desired level:

1. First, we'll add the arrays for the new row to both layers:

```
        discoveredLevel = discoveredLevel + 1
      end
      if discoveredLevel == index then
        local fixed, moving = {}, {}
        table.insert(self.Fixed, fixed)
        table.insert(self.Moving, moving)
      end
      lastLineIsContent = true
    end
```

2. Then, we'll get each character from the line and add the appropriate features to the layers. The `string.gmatch` function is used with the `for` loops to run the loops once for each piece of the string being examined that matches a given pattern, and `.` is the pattern for any single character, so `for tile in line:gmatch('.')` means repeat for each character in the variable `line`.

```
        table.insert(self.Moving, moving)
        local position = 0
        for tile in line:gmatch('.') do
          position = position + 1
          resolve[tile] (fixed, moving, position)
        end
      end
      lastLineIsContent = true
```

The last line, about `resolve[tile]`, has not actually been explained yet. Because we have several different characters, and some of them affect one layer, and some affect both, we need some way to manage choosing the appropriate processing. In most languages, this would be done with a `switch` or `case` block, which Lua doesn't offer. It could be processed in a long `if … then … elseif …then` chain, but a natural way in Lua is to use a table of functions, where the functions are stored under keys that represent their intended purposes. We'll fill in this table in a moment, because we're almost done with handling the file.

Recognizing ends of levels

When we hit a line that does not contain content, the level we were processing, if any, is over. Perform the following to recognize the ends of levels:

1. We need to make a note that we are no longer handling level content.

```
      lastLineIsContent = true
    else
      lastLineIsContent = false
    end
  end
  return self
```

2. And, if the level we were just passing through was the level that was requested, we're done. We can break out of the loop here:

```
      lastLineIsContent = true
    else
      if discoveredLevel == index then break; end
      lastLineIsContent = false
    end
```

3. Finally, before we return, we do have a sanity check to make sure that we actually found the level requested. This might not happen if, for example, we asked for the 25th level of a file with 20 levels in it.

```
      lastLineIsContent = false
    end
  end
  assert(discoveredLevel > 0, string.format([[the file "%s" was
not found to contain a level %d]], fileName, index))
  return self
end
```

Processing each tile

In the body of the `sok.load` function, we relied on a table of functions to process each tile according to its content. Some tiles affect only the fixed layer (walls), while others affect both (the player initially placed on a goal square, or a box starting on an open square).

Because the functions will be stored in a table and chosen dynamically, they all need to have the same arguments, since we do not know (and do not want to know) which one is being called. Each function will take the array representing the current row, in both the fixed and moving layers, and the column within the row being modified. So, the first function defined will affect only the fixed layer:

```
local sok = {}
local resolve = {
```

```
['#'] = function(fixed, moving, column)
  fixed[column] = '#'
end,
 }
```

This example (and all the rest) uses the extended syntax for defining arbitrary keys in a new table, `[key] = value`. While this is often used (as here) to use literal strings as keys even though they're not legal variable names, it can be used to set *any* Lua value (except nil) as the key, including tables, functions, and other complex values. Note however, that two tables with the same keys and values are still different objects, and using one as a key won't retrieve or overwrite the value associated with the other.

4. This first function affects only the fixed layer, along with the next two:

```
local resolve = {
  ['#'] = function(fixed, moving, column)
    fixed[column] = '#'
  end,
  [' '] = function(fixed, moving, column)
    fixed[column] = ' '
  end,
  ['.'] = function(fixed, moving, column)
    fixed[column] = '.'
  end,
}
```

5. The other symbols represent a movable element, either a box or the player, placed on either a plain square or a goal space. So these functions will populate both layers.

```
  ['.'] = function(fixed, moving, column)
    fixed[column] = '.'
  end,
  ['@'] = function(fixed, moving, column)
    fixed[column] = ' '
    moving[column] = '@'
  end,
  ['$'] = function(fixed, moving, column)
    fixed[column] = ' '
    moving[column] = '$'
  end,
  ['+'] = function(fixed, moving, column)
    fixed[column] = '.'
    moving[column] = '@'
  end,
```

```
    ['*'] = function(fixed, moving, column)
      fixed[column] = '.'
      moving[column] = '$'
    end,
}
```

Trailing commas in tables

Lua allows you to leave a comma (or semicolon) at the end of your tables with nothing after it, which is a good habit to get into; it makes you less likely to leave them out when adding new elements to the end of the table, especially with tables that span multiple lines.

What did we do?

We started our project by importing the beginning of the core scene that will provide gameplay. As you continue developing new apps, you'll find that you tend to produce them from a common skeleton with a little bit of adaptation, so you often won't produce the new project completely from scratch. Once we had the framework in place, we created a new module which has all the code needed to translate an encoded file into a usable level.

What else do I need to know?

The project won't load yet; try to run it and you'll get an error. The next thing we need to do is create the module that displays the `world` state on the screen. This module is already being loaded, but until it exists, you'll get an error from trying to load it.

Displaying the map contents

Right now, the game scene is dependent on another module that hasn't been created, the `world` module. This module is needed to create a visible representation of the abstract `Map` data, from the format created by the `sok.load` function.

Getting ready

The `world` module is used on the following line in `game.lua`:

```
self.World = require "world" (30, 20, lawnParty, statics, lawnParty,
movables)
```

This tells us that the `world` module needs to return a function. That function takes two numbers, the number of columns of tiles and rows of tiles that the `map` object can display at once, and several graphical arguments: an image sheet containing the tiles to use for walls, goals, and empty spaces, a table explaining which to use for each purpose, another image sheet (or in this case, the same one) containing the images to use for movable elements, and a table containing the sprite descriptions for each movable item. The function also returns some sort of object or value representing the world.

The resulting object is immediately used on two other lines:

```
group:insert(self.World)
self:addEventListener('Game', self.World)
```

We also know that the returned object has to be a display object (since it can be inserted into a group), and that it will receive `Game` events (although it is not required to act on them).

Finally, the object is used once in the game's `enterScene` method:

```
self.World:Load(self.Map)
```

So, the returned object will also need a `Load` method that takes the `Map` data as an argument. This method will arrange the elements of the `world` component to represent the specified map.

To get started, create a new `world.lua` file in your *SuperCargo* project directory and open it.

Getting on with it

We established during preparation that the module creates a function that takes certain arguments, so we'll enter the skeleton for that into the new file as follows:

```
return function(columns, rows, tileSheet, tiles, spriteSheet, sprites)
end
```

We said that the function needs to return a display object. Since it will contain many other tiles and sprites, this object should be a group:

```
return function(columns, rows, tileSheet, tiles, spriteSheet, sprites)
  local self = display.newGroup()
  return self
end
```

The returned group needs a method called `Load` that takes a `map` structure:

```
local self = display.newGroup()
function self:Load(map)
end
return self
```

Adding the content layers

We've gone on repeatedly about how the `Map` component is separated into two layers, so it makes sense to create two visual layers to parallel them:

```
local self = display.newGroup()
self.Tiles = display.newImageGroup(tileSheet)
self:insert(self.Tiles)
self.Mobs = display.newImageGroup(spriteSheet)
self:insert(self.Mobs)
function self:Load(map)
```

Optimized image groups

Since we're depending on the fact that all tiles come from a single image sheet, and all sprites also come from a single image sheet (also called a **texture atlas**), we can take advantage of Corona's image groups, which allow for higher performance when all the contents of a group can be guaranteed to use the same texture raster as their source data. While *Sokoban* is not a performance-critical game, we can experiment here with using this functionality.

Since there will be a tile in each space of the world, but we won't know which one until the map is loaded, we represent each tile with a sprite that has one sequence for each thing that tile can contain. To display the fixed part of a map, the world can just switch each tile to the sequence for the contents at that point. So that we can find each tile, we'll also store references to all of them in another two-dimensional array.

```
self:insert(self.Tiles)
self.Tiles.Row = {}
for row = 1, rows do
  self.Tiles.Row[row] = {}
  for column = 1, columns do
    local tile = display.newSprite(self.Tiles, tileSheet, tiles)
    tile:setReferencePoint( display.BottomRightReferencePoint )
    tile.x, tile.y = column * tile.width, row * tile.height
    self.Tiles.Row[row][column] = tile
  end
```

```
end
self.Mobs = display.newGroup()
```

For each row in the specified size, we create a row in the array, and enough tiles to cover every column in the row.

For the `Mobs` layer (**mob** in this case is a term from the MUD community that means **m**obile **ob**ject), we just create an array to track the positions of later sprites:

```
self:insert(self.Mobs)
self.Mobs.Row = {}
for i = 1, #self.Tiles.Row do
  table.insert(self.Mobs.Row, {})
end
function self:Load(map)
```

Loading the world with a map

Now the blank map is created, we need to define the `Load` function so that it actually copies the map content into the display. Perform the following steps to load the world with a map:

1. The first step is to make sure the `Mobs` layer is empty, in case the `map` layer was in use previously:

```
function self:Load(map)
  for i = self.Mobs.numChildren, 1, -1 do
    self.Mobs:remove(i)
  end
end
```

2. Next, we consider each tile of the world, setting the tile sprite's sequence to match the fixed layer of the map (or a blank default if the supplied map doesn't cover the whole area):

```
    self.Mobs:remove(i)
  end
  for y, row in ipairs(self.Tiles.Row) do
    for x, tile in ipairs(row) do
      tile:setSequence(map.Fixed[y] and map.Fixed[y][x] or ' ')
      local mob = map.Moving[y] and map.Moving[y][x]
    end
  end
end
return self
```

3. At the same time, we need to create sprites that represent moving objects at those locations:

```
        local mob = map.Moving[y] and map.Moving[y][x]
        if mob then
          local new = display.newSprite(self.Mobs, spriteSheet,
sprites[mob])
          new:setReferencePoint( display.
BottomRightReferencePoint)
          new.x, new.y = x * new.width, y * new.height
          self.Mobs.Row[y][x] = new
        end
      end
```

Tiling from the bottom-right corner

We set objects to be placed by their bottom-right corners, because in a system where ranges typically start at one rather than zero, this allows that corner to be placed at the index times the size of each tile and still land in the right place. For example, an object located at the tile position (3, 5) with a tile size of 16 will end up with its bottom-right corner placed at (48, 80) and its top-left at (32, 64). This means that it covers the third column and fifth row of 16 x 16 blocks starting at (0, 0).

What did we do?

Using an existing, defined data format, we created a generalized way to display the contents of a map compatible with that format.

What else do I need to know?

At this point the project should be ready to run without errors, although it will just display a single level on the screen like this and doesn't allow the user to play at all. However, one of the common strategies for programming is to develop one component, test it, and make sure it works before adding new features. Now we know we have a working map and can focus on adding gameplay.

If your project doesn't work, particularly check that you modified the `main.lua` file to launch the `game` scene rather than `scenetemplate`, and that you're passing it the `File` and `Level` fields in its `params` table.

Adding the interface

Now that the game displays a level correctly, we need to start adding the ability to communicate with it.

Getting ready

We imported a pre-packaged `interface.lua` file from `version 1`, but it doesn't do anything. Actually, it does one thing: it returns a blank group, which fills in the space that the rest of the game needs occupied to work, but doesn't actually send or respond to any events.

This practice, when programming complex systems, of leaving a minimal dummy placeholder for a larger chunk of code in place until the actual implementation can be created is called making a `stub`.

Open the `interface.lua` file and remove the starting placeholder that says:

```
return display.newGroup()
```

Getting on with it

We'll expand the interface group with the required behaviors:

1. The interface will still be represented by a `display` group, so we'll break it out into a variable so that we can add to it before we return it:

```
return function(game)
  local self = display.newGroup()
  return self
end
```

2. The design says that the interface will display a move count in the upper-left, so we'll create a text object, which starts out displaying `0`:

```
  local self = display.newGroup()
  local counter = display.newText(self, "0", 10, 10, native.
systemFont, 24)
  counter:setReferencePoint(display.CenterLeftReferencePoint)
  return self
end
```

3. When a `Move` event happens on the `game` object, this count needs to be updated. The following pattern to adjust the text and keep it aligned should be familiar:

```
counter:setReferencePoint(display.CenterLeftReferencePoint)
function counter:Move(event)
  if event.count then
    local x, y = self.x, self.y
    self.text = event.count
    self:setReferencePoint( display.CenterLeftReferencePoint)
    self.x, self.y = x, y
  end
end
game:addEventListener('Move', counter)
return self
```

   In other words, the counter listens for the `Move` events on the game; if the event has an associated count of moves taken, the counter updates itself to display that count.

4. It was also stated that the counter fades if the game is left undisturbed. We need to know what to monitor for touch events that would wake it up; while we could use `Runtime`, that creates problems when the scene stops being active (we'll discuss that problem in more detail in a later project). The interface will collect all its `touch` and `tap` events from the scene's associated group, which collects all touches on its children (such as the `world` scene) that aren't handled by those children.

```
return function(game)
      local self = display.newGroup()
  local target = game.view
  local counter = display.newText(self, "0", 10, 10, native.
systemFont, 24)
```

5. Whenever a touch on this target is detected, the counter needs to cancel any previous fading it had planned, and start a new timer. We can make the timer start fading and the fade over time simple by using a transition with an initial delay, which is reusable:

```
local fade = {delay = 5000, time = 750; alpha = 0}
return function(game)
```

6. The `touch` handler becomes fairly easy to write at this point; we need to track the transition for the `fade` event, so it can be cancelled, which is easily done as a property of the text object:

```
    self.x, self.y = x, y
  end
end
function counter:touch(event)
  if self.Fade then
```

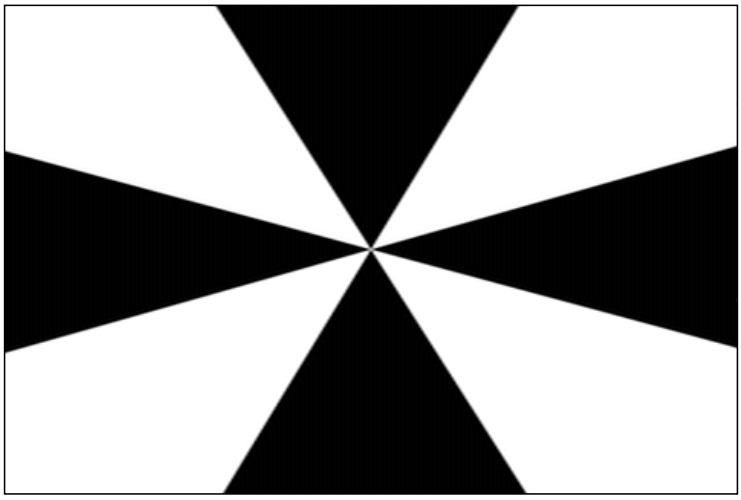
```
    transition.cancel(self.Fade)
  end
  self.alpha = 1.0
  self.isVisible = true
  self.Fade = transition.to(self, fade)
end
target:addEventListener('touch', counter)
game:addEventListener('Move', counter)
```

7. Finally, we need to start the timer running when the interface is created, or the counter will sit on the screen at the game start without fading until the player touches the screen:

```
  self.Fade = transition.to(self, fade)
end
counter:touch()
target:addEventListener('touch', counter)
```

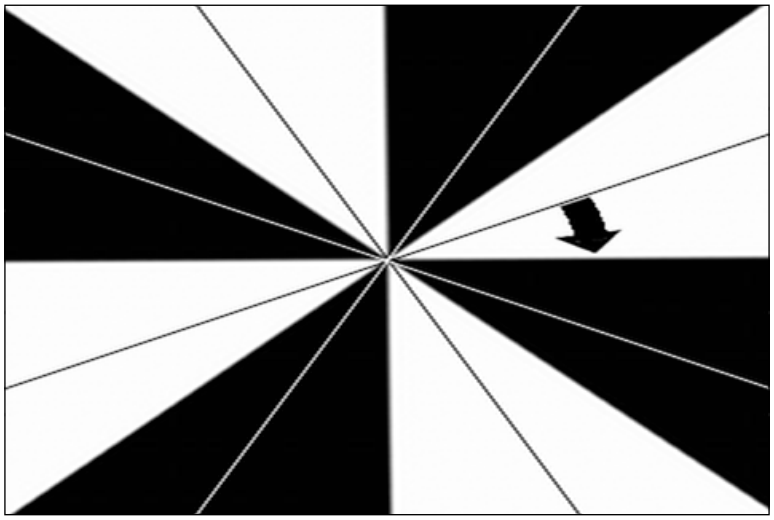
Creating the Move requests

The last element of the interface is not complicated code, but it takes some real math to design properly. We need to identify taps as being in one of the four compass directions compared to the screen. To reduce user mishaps, we will discard taps that happen too near a boundary between one direction and another.



Black regions indicate where touches will move the character

We have two challenges here: recognizing whether a touch is in one of the active zones or not, and "snapping" the directions being output to the exact compass points. We can get the angle of the touch with the `math.atan2` function, but the tough part is the east quadrant. The polar axis, the point where the zero angle lies on graphs, is right in the middle of it. Angles that should be registered for the eastern direction range from 0-π/8 (Lua's trigonometric functions all return results in radians), but also from 15π/8-0. While this problem could be solved with a long `if/elseif` statement, we can solve the whole problem fairly simply by just adding π/8 (one-sixteenth of a circle), moving the angles under consideration to line up with the zero line.



At this point, solving both questions becomes much easier. We can slice the circle into eight wedges, and determine which wedge the specified angle falls into by dividing the angle by one-eighth of a circle, π/4. You can clip the fractional part of the wedge off with `math.floor()` and normalize it into a range (eliminating negative angles or angles over a full circle) by taking the result modulus `8` (that is, dividing by eight and keeping only the remainder).

In this way, a fractional number from 0 to 2π is converted into an integer from zero to seven. Because we're only interested in the wedges zero, two, four, and six, we can compare the wedge index modulus `2` to `0` to determine if it's a valid touch, and then multiply the wedge number by an eighth of a circle, π/4, to obtain the final direction, pointing straight right, down, left, or up.

```
local function checkMove(self, event)
    local direction = math.atan2(event.y - display.contentCenterY,
event.x - display.contentCenterX) + math.pi / 8
    direction = math.floor(direction * 4 / math.pi) % 8
```

```
    if direction % 2 == 0 then
      self:dispatchEvent{name = 'Move'; action = 'attempt', direction
= direction * math.pi / 4 }
    end
  return true
end
return function(game)
  local self = display.newGroup()
```

This new function just has to be attached to the interface to handle touches on the target using the following code:

```
  local target = game.view
  self.tap = checkMove
  target:addEventListener('tap', self)
  local counter = display.newText(self, "0", 10, 10, native.
systemFont, 24)
```

The `dispatchEvent` call triggers an event targeted on the interface that indicates that the player tried to move in a particular direction. This event has no clue about whether the attempted move is legal or will produce any results. It only says that the player tried to move, and which way.

What did we do?

We fleshed out the interface so that the user knows how deep they are into the game, and added a little polish so the UI feels more professional. We also created a layer that takes a complex event and reduces it to just the information the game needs to know.

Making the game playable

The interface registers touches as commands now, but the game isn't receiving them yet. So it will still appear dead as a doornail, except insofar as touching the screen will make the move counter (which is stuck at zero) stop fading or reappear.

Getting ready

Copy the file `map.lua` from the `version 2` folder into your `Supercargo` directory, and modify `game.lua` by replacing the line that says the following:

```
self.Map = sok.load(event.params.File, event.params.Level)
```

So that it says the following instead

```
  self.Map = require "map" (sok.load(event.params.File, event.params.
Level))
```

The function in the `map` module modifies `map` as returned by `sok.load` by adding some methods to it that help resolve rules questions about the `map` object. One returns the column and row where the player is currently located in the `map` object, one checks the map to see if the game has been won (all boxes are on goal spaces), and one checks whether there is a potential obstacle to player movement (a wall or box) at a given space, and what it is (either `#`, `$`, or nil). We'll want this function to determine if proposed moves are legal.

Getting on with it

To make the game actually process move inputs, it needs to register for the `Move` events sent to the interface, in its own `createScene` responder:

```
  self.UI = require "interface" (self)
  group:insert(self.UI)
  self.UI:addEventListener('Move', self)
end
```

Handling the Move inputs

We need to let the scene respond in some way when `Move` events occur. The following are the steps for handling the `Move` inputs:

1. The default will be able to assume that the move fails, unless we determine that it is legal; we'll modify our `template` event as we discover the actual circumstances.

```
function scene:Move(event)
  if event.action == 'attempt' then
    local move = {name = 'Move'; action = 'blocked', direction =
event.direction}
    self:dispatchEvent(move)
  end
end
-- Called when the scene's view does not exist:
function scene:createScene( event )
```

2. The game will need to check the legality of the move, modify its state according to the move's outcome, and report the outcome of the move. The `Move` event contains the desired direction of movement, so we'll have to convert this to a destination tile using the player's current location:

```
  if event.action == 'attempt' then
    local x, y = self.Map:FindPlayer()
```

```
    local dX, dY = round(math.cos(event.direction)), round(math.
sin(event.direction))
    local targetX, targetY = x + dX, y + dY
    local move = {name = 'Move'; action = 'blocked', direction =
event.direction}
```

3. The `if` statement about the event type is mostly a sanity check at the moment, but will become important when we add `undo`. The `round` function is a trivial function that rounds a number toward the nearest integer, rather than strictly up or strictly down; you can add it to `game.lua` right above the `scene:Move` method:

```
local function round(x)
  return math.floor(x + 0.5)
end
function scene:Move(event)
```

4. Now, we know what space the player is trying to move his character to, so we can ask the map whether that space is clear:

```
    local targetX, targetY = x + dX, y + dY
    local obstacle = self.Map:Obstacle(targetX, targetY)
    local move = {name = 'Move'; action = 'blocked', direction =
event.direction}
```

5. We can deal with the simple case first; the space is empty and we can move into it freely:

```
    local move = {name = 'Move'; action = 'blocked', direction =
event.direction}
    if not obstacle then
      self.Map.Moving[y][x], self.Map.Moving[targetY][targetX] =
nil, self.Map.Moving[y][x]
      if #move == 0 then
        move.action = 'step'
      end
      table.insert(move, 1, {x = x, y = y})
    end
    self:dispatchEvent(move)
```

Lua supports multiple assignment, and it can be a lot more concise when moving values from one variable or location to another, especially if they're being swapped. The part where we check the length of the move event and insert something into it needs a bit more of an explanation.

Tables in Lua can contain values under any kinds of keys, including multiple kinds in the same table. When we store values under string keys that name the relationship of the value to the table, we call the table, a *record*; when we store things under consecutive integer keys, we call it an *array*. In this case, a single event table is being used as both at once.

I like to use the array portion of an event to indicate that things are direct objects of the event action, especially when there can be more than one. So if a move event represents a successful move, the array portion contains all the things that are moving, denoted by their original coordinates; the first one is assumed to be the player (who always moves on any successful move event) and any others are boxes (a maximum of one under standard *Sokoban* rules).

6. If anything is moving, we also want to include the horizontal and vertical movement, since we've already calculated them:

```
      table.insert(move, 1, {x = x, y = y})
    end
    if #move > 0 then
      move.x, move.y = dX, dY
    end
    self:dispatchEvent(move)
```

7. Now, if there *is* an obstacle, it doesn't automatically mean the move fails. Boxes can be pushed as long as there is an empty space directly beyond them. This check goes before checking whether there is free space, since if the box can move, we'll move it over and leave free space that the player can move into:

```
    local move = {name = 'Move'; action = 'blocked', direction =
event.direction}
    if obstacle == '$' then
      local finalX, finalY = targetX + dX, targetY + dY
    end
    if not obstacle then
```

8. This says that as far as the box is concerned, we're looking at the space from the same distance from the box in our way, as the box was from the guy pushing it. We need to know if that space is free.

```
    if obstacle == '$' then
      local finalX, finalY = targetX + dX, targetY + dY
      if not self.Map:Obstacle(finalX, finalY) then
        self.Map.Moving[targetY][targetX], self.Map.Moving[finalY]
[finalX] = nil, self.Map.Moving[targetY][targetX]
      end
    end
```

9. This moves the box into the space next to it, in the direction in which the character pushed. Next, we add the box to the event as something that is moving:

```
      if not self.Map:Obstacle(finalX, finalY) then
        self.Map.Moving[targetY][targetX], self.Map.Moving[finalY]
[finalX] = nil, self.Map.Moving[targetY][targetX]
        table.insert(move, 1, {x = targetX, y = targetY})
      end
```

10. As we described previously for the player, we're saving the box's current position in the event. We also want to mark the event as moving a box, because that makes it a substantive change to the game state.

```
    table.insert(move, 1, {x = targetX, y = targetY})
      move.action = 'push'
  end
```

11. To complete the push, we indicate that there is no obstacle left in the space, so that the player will also move. Note that if the box can't move, the obstacle variable won't be cleared, so the player won't move either.

```
        move.action = 'push'
        obstacle = nil
      end
```

    There are only two jobs left. We need to include a move count, so that the counter on the screen can update, and then we need to end the game if the win condition has been met.

12. We'll use a new property of the game scene, `MoveCount`, to track how many pushes have taken place, so we'll need to jump briefly to the scene's `enterScene` method, so we can declare that any newly started game has used zero moves.

```
  self.Map = require "map" (sok.load(event.params.File, event.
params.Level))
  self.MoveCount = 0
  self.World:Load(self.Map)
```

13. Returning to the `Move` handler, we need to increase the `MoveCount` value whenever a box gets moved, since simple player steps can't cause a win.

```
      move.x, move.y = dX, dY
    end
    if move.action == 'push' then
      self.MoveCount = self.MoveCount + 1
    end
    move.count = self.MoveCount
self:dispatchEvent(move)
```

14. Finally, after the `move` event is finished, we need to check whether the game's condition was changed to a `win` condition. Since the game state wasn't actually changed unless the `move` event was `push`, we'll repeat the check we just made:

```
    self:dispatchEvent(move)
     if move.action == 'push' and self.Map:CheckWin() then
       self:dispatchEvent{name = 'Game'; action = 'stop', state =
self.Map, Game = self}
     end
   end
```

    This means that if a `win` condition is reached, we'll send out an event saying that the game is over. This allows things like the `world` object to clean up their association with the game.

15. To see something actually happen, the game itself needs to respond to this event as well; for now, we'll just close the app when the level is over:

```
function scene:Game(event)
  if event.action == 'stop' then
    os.exit()
  end
end
-- Called when the scene's view does not exist:
function scene:createScene( event )
```

16. And to actually receive the event, we should register it as well:

```
    os.exit()
  end
end
self:addEventListener('Game', self)
--
 Called when the scene's view does not exist:
```

Displaying the effects of moves

Right now, we could technically play through the level, and win it, but we'd be playing completely blind. We'd never see a thing happen on the screen. The moves take place in the game, but nothing fixes the world to reflect that. We want the world to listen for the `Move` events on the game (the `Move` events on the interface are only indications of player intention, not actual changes, so the world isn't interested in them). Perform the following steps:

1. The game doesn't know that the world is interested in its `Move` events, but it does make sure that the `world` object gets its `start` and `stop` events, so we'll have the world respond to those by registering itself appropriately. Open up `world.lua` and add a new function to the `world` object:

```
  function self:Game(event)
    if event.action == 'start' then
```

```
    event.Game:addEventListener('Move', self)
  elseif event.action == 'stop' then
    event.Game:removeEventListener('Move', self)
  end
end
return self
```

2. The world isn't interested in the `Move` events if there's no game going on (and not being registered when the game scene isn't active helps avoid some registration issues). This ensures that once the game has started, and only for the duration of the game, the world will receive the game's `Move` outcome events. But it also needs to respond to them:

```
function self:Move(event)
  if event.action == 'step' or event.action == 'push' then
  end
end
function self:Game(event)
```

3. We aren't interested in blocked move events, since we don't have to update anything. Next we make a note of which way things are moving (In *Sokoban*, if multiple things move in a single action, they always move in the same direction):

```
  if event.action == 'step' or event.action == 'push' then
    local dX, dY = event.x, event.y
  end
```

4. The next part is a little strange:

```
    local dX, dY = event.x, event.y
    local moving = {}
    for _, coords in ipairs(event) do
      moving[ self.Mobs.Row[coords.y][coords.x] ] = coords
      self.Mobs.Row[coords.y][coords.x] = nil
    end
  end
```

This is working around a potential pitfall. Because the first thing in the list is the player, and if a box is moving, the player is moving into the space where the box currently is. If we simply moved the reference to the player sprite into the space where it's going, it would overwrite the reference to the box sprite being pushed, and we wouldn't be able to find the box to move it.

5. There are a few ways to work around the issue, but this is the safest; we make a temporary table, and copy all the things that are moving into that table with their original locations as the associated values. Then, we take them out of the world's list of movable things.

```
        self.Mobs.Row[coords.y][coords.x] = nil
      end
      for mob, coords in pairs(moving) do
        self.Mobs.Row[coords.y + dY][coords.x + dX] = mob
        mob.x, mob.y = mob.x + mob.width * dX, mob.y + mob.height
* dY
      end
  end
```

Then what we do is go through the temporary table of things that are moving. Since each sprite that we are moving was matched in the table with its current coordinates, we put it back into the `Mobs` layer at its new position, obtained by adding the movement directions to those current coordinates. Then we move the actual sprite, and not just the tile position it is recorded at being in, by the appropriate amount.

What did we do?

We made the game process user input into actual moves, enforcing the game rules, and we made the world display those moves so the player can see what is going on.

At this point, you should be able to play through the level! Sit back for a few moments, build the project for your device, and enjoy the fruits of your work.

Adding the Shell component

Now that we have one playable level, we need to expand this app to allow the player some choice. The included file comes with 90 levels and limiting the player to only the first one seems unfortunate.

Getting ready

Start by copying the `images/splash.png` and `menu.lua` files from the `version 3` folder of the project pack. This gives you a working foundation for a shell, although it is not very interactive. To add the `Shell` component, perform the following steps:

1. Open `main.lua` and fix the `storyboard.gotoScene` call there to load the `menu` scene rather than the `game` scene.
2. Next, open `game.lua`, find the `scene:Game` function that dictates what should happen when a game ends, and replace the call to `os.exit()` with `storyboard.gotoScene( "menu" )`.

3. Test the app to make sure it launches to the menu screen. The **Go!** button starts the familiar level, and completing the level returns to the menu screen.

Getting on with it

When the user taps the place where the currently selected level is displayed, the app should pop up a list of available levels to choose from. For the time being, this will stay simple: it will just say `1` for the first level, and so on. To do this, the menu needs to determine how many levels are in the file in question, display a list, and get the user's selection back from that list.

Counting levels

Fortunately, we have a library which already does something related. We'll add a new function to the `sok` module, `sok.count`, which will return the number of levels found in a given file. Perform the following steps to count levels:

1. The `sok.count` function has a lot in common with `sok.load`. In fact, `sok.count` was originally created by copying the function body, changing the name, and stripping out the code that actually built the map. It just lets `discoveredLevel` increase until it runs out of file, and returns the resulting number. The following is the finished function:

```
function sok.count(fileName)
  local discoveredLevel, lastLineIsContent = 0, false
  for line in io.lines(fileName) do
    if line:match("^[# .@+$*]+$") then
      if not lastLineIsContent then
        discoveredLevel = discoveredLevel + 1
      end
      lastLineIsContent = true
    else
      lastLineIsContent = false
    end
  end
  return discoveredLevel
end
```

2. You can see how this is based on the same framework. Now, we need to load this module in `menu.lua` so that the menu can inform the level selection of how many levels are available:

```
local scene = storyboard.newScene()
local sok = require "sok"
scene.File = system.pathForFile "levels/default.txt"
```

3. Since the `pickLevel` module will always come back to the `menu` scene, we'll implement it as an overlay scene, much the way we did with the initials entry module in Bat Swat. Add a `tap` listener to show this overlay when the user taps the current level:

```
  self.LevelSelect.Display = display.newText( self.LevelSelect,
"1", 6, 5, native.systemFont, 20)
  self.LevelSelect:addEventListener('tap',
    function(event)
    end
  )
  self.Launch = display.newGroup()
```

4. We'll give the selection screen three pieces of information that it needs to do its job: which object should the user be notified with when the user makes a choice (by receiving a `SelectLevel` event), which levels are available for selection (for convenience, this can just be a number, and the list will autopopulate), and which level was currently selected (so that if the player finished 68 and wants to switch to 69, he/she doesn't have to scroll all the way back down from 1).

```
  self.LevelSelect:addEventListener('tap',
    function(event)
      local parameters = {
        Target = self;
        Selection = tonumber( self.LevelSelect.Display.text);
        Range = sok.count(self.File);
      }
    end
  )
```

5. Now that it's ready, we'll invoke the selection screen with those arguments when the selection box is tapped:

```
        Range = sok.count(self.File);
      }
      storyboard.showOverlay( 'pickLevel', {isModal = true; params
= parameters})
    end
  )
```

6. Since we supply the scene object itself as the `Target` object for the `LevelSelect` events, something needs to listen to it in order to capture the selection information. We can use the selection text, which is already used to store the current selection:

```
  function self.LevelSelect.Display:SelectLevel(event)
  self.text = event.Level
```

```
end
self:addEventListener('SelectLevel', self.LevelSelect.Display)
self.Launch = display.newGroup()
```

So to write the `pickLevel` module, we know that it has to function as a scene, and we also know that it receives a `Target` object to which it should send the `SelectLevel` events containing a `Level` field. We know that it needs to offer a choice among some number of levels, and that it should start with one specified level pre-selected. We'll accomplish all this with a tableView, one of Corona's built-in UI templates.

Building the selection screen

Create a copy of `scenetemplate.lua` named `pickLevel.lua` and open it. The following are the steps for building the selection screen:

1. At the top, load the `widget` library so that we can create a tableView:

```
local scene = storyboard.newScene()
local widget = require "widget"
-- Called when the scene's view does not exist:
function scene:createScene( event )
```

2. In the `createScene` body, make a tableView. Since it's fine for it to cover the whole screen, we can use defaults for most of the inputs and just supply a couple of options:

```
function scene:createScene( event )
  local group = self.view
  self.Display = widget.newTableView{
    id = "level_selection",
    bgColor = {0x40, 0x40, 0x40, 0x99},
  }
  group:insert(self.Display)
end
```

3. The real work will come once the `enterScene` module has been called and we know that we have the right set of parameters.

Although `createScene` also receives the parameters passed to `storyboard.gotoScene`, be careful of relying on them in `createScene`. If your scene is loaded again without being purged first, the `createScene` event will not be fired again, and you'll enter the scene still using values that were passed the first time the scene was entered.

4. We'll save the event recipient in the `scene` object so that it can be referenced easily.

```
function scene:enterScene( event )
  self.Target = event.params.Target
end
```

5. We need to add rows to the table that represent the different levels available. This is easiest to do in a loop.

```
  self.Target = event.params.Target
  local exploreAll = ipairs
  local labels = {}
  for i, text in exploreAll(event.params.Range) do
    labels[i] = text
    self.Display:insertRow{
      height = display.contentHeight * 0.10,
    }
  end
end
```

   This populates a table with strings obtained from a table passed in. But this is mostly future-proofing, against the day we get level files with descriptions included or something similar; we've already passed a number for that value, instead. That's why we use an intermediary variable to store `ipairs` instead of just calling `ipairs` directly for the `for` loop.

6. We need to create a loop driver that just iterates up to a number, returning indices and strings the same way `ipairs` would.

```
  local exploreAll = ipairs
  if tonumber(event.params.Range) then
    exploreAll = countTo
  end
  local labels = {}
```

7. We'll add the `countTo` iterator function and its supporting `countUp` function at the top of the file:

```
local widget = require "widget"
local function countUp(max, n)
  n = n + 1
  if n <= max then
    return n, tostring(n)
  end
end
local function countTo(n)
  return countUp, n, 0
end
-- Called when the scene's view does not exist:
```

Lua's `for…in…do` structure is called the **generic for** because you can completely control how it generates new values. It takes three things: a generator function, a state variable that stays constant until the loop is over, and a starting position variable, allowing you considerable control. Functions such as `pairs`, `ipairs`, and `string.gmatch` fill in these three values for you when you call them. We'll look more closely at custom iterators in *Project 9*, *Into the Woods – Computer Navigation of Environments*.

Presenting the table

Corona's tableView provides a few default aspects of appearance; blocks of background color with lines between rows, and so on. But since the contents of a table are very free-form, it's up to your code to explain how they will be presented. The following are the steps for presenting the table:

1. The tableView has two different aspects on which it requires your assistance; these are presenting the table's data, and processing user input.

```
local labels = {}
local function displayLevel(event)
end
local function selectLevel(event)
end
for i, text in exploreAll(event.params.Range) do
  labels[i] = text
  self.Display:insertRow{
    height = display.contentHeight * 0.10,
    onRender = displayLevel,
    onEvent = selectLevel,
  }
end
```

   The specified functions receive events containing the particulars, such as what groups to draw in and which row in the table the event is for.

2. The tableView does part of the display according to defaults unless you override it, creating white rows with thin lines between them. That's fine for a practice project. What we do need to add is the text for the appropriate level:

```
local function displayLevel(event)
  local row, output = event.target, event.view
  display.newText(output, labels[event.index], 12, row.height *
0.125, native.systemFont, row.height * 0.75)
    :setTextColor(0x00)
end
```

3. We get the height of the row from the original row creation (10 rows on the screen at once), the group to add the text to from the event, and the number of rows within the table from the event as well. We use the shared `labels` table to get the text from the loop that created the rows. Then, we just need to process touches on the rows:

```
  local function selectLevel(event)
    local row, output = event.target, event.view
    if event.phase == 'press' then
      output.background:setFillColor(0x99)
    elseif event.phase == 'release' then
      self.Target:dispatchEvent{name = 'SelectLevel'; Level =
event.index}
      storyboard.hideOverlay()
    end
  end
```

   Row touch responders get four main kinds of touches. We're not interested in horizontal swipes; we respond to press actions just to highlight the choice the user is hitting. A release is treated as a selection, sending the required event back to the menu scene and hiding the selection process.

4. Now that the scene can create the table correctly on pressing *Enter*, we just need to start out by scrolling to the desired part of the table:

```
      onEvent = selectLevel,
    }
  end
  self.Display:scrollToIndex(event.params.Selection)
end
```

5. There is one last piece of cleanup to take care of. Since the `Display` text object was registered as a listener on the scene, if the scene's view is purged, the text object will be destroyed but will remain as a listener, causing problems the next time the scene is reconstructed. We can clean this up in the `destroyScene` handler for the menu module:

```
-- Called prior to the removal of scene's "view" (display group)
function scene:destroyScene( event )
  self:removeEventListener('SelectLevel', self.LevelSelect.
Display)
end
```

What did we do?

We brought in a fairly straightforward static scene and expanded it with an outside module, so that the tasks of starting a game and selecting one can be kept separate. This also makes it easier to change either the menu or the selection process later.

Supporting Undo

In puzzle games, it's important to consider alternate solutions. It's especially important in games where some moves cannot be reversed, such as pushing a box into a corner. We'll use the game history we've accumulated to step backward and reverse our most recent move when the user shakes the device.

Getting on with it

Our design says that the user should be able to undo their last move by shaking the device.

Recognizing the Undo requests

Shaking is recognized by looking at the `accelerometer` events. These events are dispatched only to the global `Runtime` target; our UI will need to listen to this target, and stop listening when its scene is not active, but ideally we don't want the game scene to know that a submodule needs to be connected to `Runtime` or disconnected later. Perform the following steps to recognize the `Undo` requests:

1. Since the interface for the game scene is generated in the `createScene` event, the interface can register and unregister itself by listening to the `Scene` object for the `enterScene` and `exitScene` events. We'll add the functions to listen or stop listening to `Runtime` first, in `interface.lua`:

```
local function engage(self, event)
  Runtime:addEventListener('accelerometer', self)
end
local function disengage(self, event)
  Runtime:removeEventListener('accelerometer', self)
end
return function(game)
```

2. Then we just need to attach the following functions to the interface when it is constructed, and start listening for the right scene events:

```
  target:addEventListener('tap', self)
  self.enterScene = engage
  game:addEventListener('enterScene', self)
  self.exitScene = disengage
```

```
  game:addEventListener('exitScene', self)
  local counter = display.newText(self, "0", 10, 10, native.
systemFont, 24)
```

3. And finally, since the interface object itself is being registered as the listener for the `accelerometer` events, it needs an appropriate method:

```
  game:addEventListener('exitScene', self)
  function self:accelerometer(event)
    if event.isShake then
      self:dispatchEvent{name = 'Move'; action = 'undo'}
    end
  end
  local counter = display.newText(self, "0", 10, 10, native.
systemFont, 24)
```

Fortunately, we don't have to do any complex tracking of `accelerometer` data to recognize whether the user is shaking the device; Corona uses routines provided by the host OS to determine that for us.

Saving move history and backing moves out

So now the interface can dispatch the `undo` events for `Move` when the device is shaken. The `Game` object is already listening for the `Move` events on the interface, so we need to add some recognition to the existing routine. Open `game.lua` and find the `scene:Move` function.

```
    if move.action == 'push' and checkWin(self.Map) then
      self:dispatchEvent{name = 'Game'; action = 'stop', state = self.
Map, Game = self}
    end
  elseif event.action == 'undo' then
  end
end
```

The `Move` function currently only acts when the requested action is an attempt to move. When that's not the case, we can now check whether the `Move` was an `undo` request, instead.

The question then becomes how to restore the game back to its state before a given move. Fortunately, each `Move` event on the game that's either a `push` or `step` event contains the positions of the moving elements before the move was completed. So we can actually save these events themselves as a way of tracking the game history (specifically, the `push` events, since player steps don't meaningfully advance the game). Perform the following steps for saving move history and backing moves out:

1. Right now, the game scene uses a number to count how many `push` events have taken place. But since we're going to be saving them in chronological order in an array, we can use the length of that array to know how many moves have taken place instead. Find the `scene:enterScene` code in `game.lua` and change the following line:

```
  self.Map = require "map" (sok.load(event.params.File, event.
params.Level))
  self.MoveCount = 0
  self.World:Load(self.Map)
```

   So that it reads the following:

```
  self.Map = require "map" (sok.load(event.params.File, event.
params.Level))
  self.History = {}
  self.World:Load(self.Map)
```

2. We can then match the other code in `scene:Move` that formerly relied on `MoveCount` to this new system:

```
    if move.action == 'push' then
      table.insert(self.History, move)
    end
    move.count = #self.History
    self:dispatchEvent(move)
```

3. Now that our pre-existing code is working on the new system, we can use the most recent `Move` event in the history as a way of resetting the `Move` action:

```
  elseif event.action == 'undo' then
    local lastMove = table.remove(self.History)
    if lastMove then
    end
  end
```

   The `if` block just ensures that we don't attempt to `undo` the beginning of the game.

4. Since we don't save `step` events in the history, we can't guarantee that the player is still in the position it was in immediately after the move being undone. Although the box pushed will be in the same position as it was in immediately after the move being undone, since this is the most recent push. So we get the player's current position from the map, clear that position, and set the player's position before the move being undone as his/her current position:

```
    if lastMove then
      local column, row = self.Map:FindPlayer()
      local realm = self.Map.Moving
      realm[row][column] = nil
      realm[lastMove[1].y][lastMove[1].x] = '@'
    end
```

5. The local variable realm is just used to shorten our code and eliminate a lot of tedious repeating of table lookups. Then, we can fetch the box that was moved and restore it from the position it was moved to.

```
    realm[lastMove[1].y][lastMove[1].x] = '@'
    local xFinal, yFinal = lastMove[2].x + lastMove.x,
lastMove[2].y + lastMove.y
    realm[lastMove[2].y][lastMove[2].x] = realm[yFinal][xFinal]
    realm[yFinal][xFinal] = nil
  end
```

6. Finally, we reload the world to match the current state of the `Map` entity (this is much easier than trying to regress it) and issue an event with the revised move count to force the interface to update.

```
    realm[yFinal][xFinal] = nil
    self.World:Load(self.Map)
    self:dispatchEvent{name='Move'; count = #self.History}
  end
```

At this point, undo should work. You can load the app to your device and test it the real way, or it will simulate device shakes although the simulator doesn't supply most `accelerometer` input.

What did we do?

We replaced a simple move count with a series of `Move` actions that we can reverse. We didn't have to generate any new data structures to hold the history as the events we already used to process gameplay have all the needed info!

Preserving game history

Finally, we descend on the task that we set on this project to solve. The rest of the game now meets its requirements and it is up to us to make sure that someone's partially solved game is not lost due to circumstances.

Getting ready

Copy the `save.lua` file from the `version 5` folder in the project pack in your directory. This module will do the actual file writing and editing.

We'll use a file in the `Documents` directory to store the game history so that it can be reloaded after the app is restarted. Now, when the menu launches a level, it will first check whether that file is present and contains history. If so, it will ask the user whether to continue from it, or delete it and start the level over. To avoid name collisions, the history file for a level will be the number of that level, followed by a period and the name of the level file, for example, `13.default.txt`.

Getting on with it

The game history consists of three main elements—managing user choices about using history, saving actual history, and linking the `save` process to the game in progress.

Controlling history selection

To get started, open up the `menu.lua` file. Perform the following steps to control history selection:

1. We'll use a more involved function now to load the selected level, so replace the line that starts `self.Launch:addEventListener` with the following code:

```
  display.newText(self.Launch, "Go!", 16, 5, native.systemFont,
20)
  local function launch(event)
  end
  self.Launch:addEventListener('tap', launch)
```

2. We'll save some reusable values in local variables before checking whether the `history` file exists for the level being started:

```
  local function launch(event)
    local level = tonumber(self.LevelSelect.Display.text)
    local history = string.format("%d.%s", level, self.
File:match("[^/\\]+$"))
    local parameters = {
      File = self.File;
      Level = level;
      Path = system.pathForFile(history, system.
DocumentsDirectory)
    }
  end
```

3. Next, we'll attempt to open the designated file to see if it exists or not.

```
      Path = system.pathForFile(history, system.
DocumentsDirectory)
    }
    local existingHistory = io.open(parameters.Path, 'r')
  end
```

4. If the file is empty, we won't bother the user with resuming a game that never made any progress, as if there were no history, and just start the game:

```
    local existingHistory = io.open(parameters.Path, 'r')
    if existingHistory and existingHistory:seek('end') > 0 then
    else
      storyboard.gotoScene( "game", {params = parameters} )
    end
  end
```

5. If the file does exist, it can't be deleted if it's still open.

```
    if existingHistory and existingHistory:seek('end') > 0 then
      existingHistory:close()
    else
```

6. Rather than taking the trouble of creating a new scene, we'll be using the platform's native alert mechanism to ask the users what they want to do:

```
    if existingHistory and existingHistory:seek('end') > 0 then
      existingHistory:close()
      local function proceed(event)
      end
      native.showAlert("Game in progress", "Do you want to resume
or delete this game?", {"Resume", "Reset", "Cancel"}, proceed)
    else
```

7. The `showAlert` function will eventually dispatch an event to the specified listener, `proceed`, that indicates whether the user picked the first, second, or third option. We'll move on to the game scene as long as the user didn't select **Cancel**:

```
      local function proceed(event)
        if event.index < 3 then
          storyboard.gotoScene( "game", {params = parameters} )
        end
      end
```

8. But first, if the user selected the **Reset** option, we'll clear the `history` file:

```
      local function proceed(event)
        if event.index == 2 then
          os.remove(parameters.Path)
        end
        if event.index < 3 then
```

Linking history to the game when loading

Now the menu scene is confirming the user choice correctly, as well as supplying the game scene with the location it will use to save or load history info. The history will consist of all the `Move` attempt events sent to the interface that actually resulted in a legal move; each event will be encoded in text using JSON format, which can be easily encoded and decoded using Corona libraries. So on entering the scene, we'll recover all such events from the file, in order, and send them to the UI as if they had been triggered by tap events. We'll do this before loading the world with the `map` data or posting the game's `start` event, so the world doesn't spend time keeping up with displaying moves that will just be overwritten an instant later.

JSON is **JavaScript Object Notation**, a common format for recording objects with named properties that looks a lot like a Lua table constructor.

Perform the following steps to link history to the game when loading:

1. First, at the beginning of `game.lua`, load the JSON library:

```
local sok = require "sok"
local json = require "json"
local storyboard = require( "storyboard" )
```

2. Then, find the `enterScene` function:

```
self.History = {}
self.Path = event.params.Path
local historyFile = io.open(self.Path, 'r')
self.World:Load(self.Map)
```

3. If the file doesn't exist, we'll create it empty, and leave it closed to avoid any problems with the `save` module:

```
local historyFile = io.open(self.Path, 'r')
if historyFile then
else
  io.open(self.Path, 'w')
    :close()
end
self.World:Load(self.Map)
```

4. But if it does exist, we'll read out each line of text, use the JSON library to convert it back into a `move` event and dispatch it to the interface, just like the interface itself would, but without the `tap` event. The scene's own event processing will then update the game with the results of the move.

```
if historyFile then
  for line in historyFile:lines() do
    self.UI:dispatchEvent(json.decode(line))
  end
  historyFile:close()
else
```

5. To complete the `save` system, whether we loaded any history or not, we engage the save module that we imported earlier and register it to clean itself up when the scene is exited as shown in the following snippet:

```
  end
  self:addEventListener('exitScene', require "save" (self.Path,
self.UI, self))
  self.World:Load(self.Map)
```

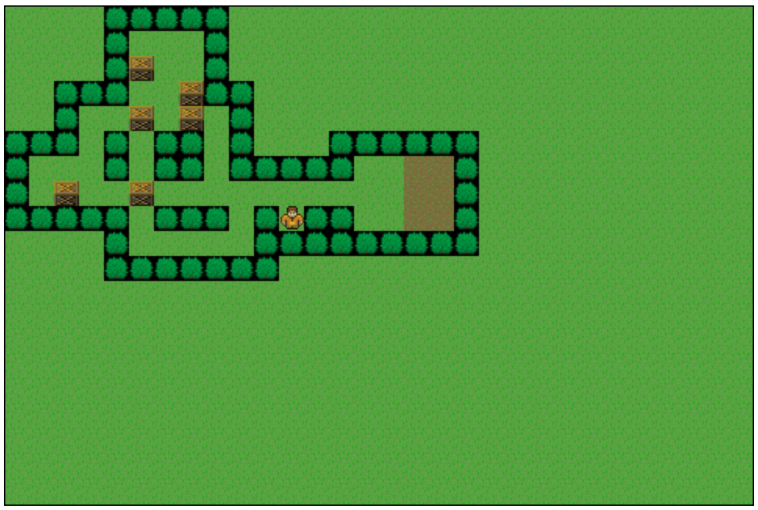
The `save` module makes a point of only saving move attempts that result in legal moves. This means that it needs access to the source of move attempts (the `interface` object), but also to the source of events that confirm those moves (the game scene). It also needs to know the path it will use for the file. For convenience, the `save` module returns a function that turns it off, so that the game scene doesn't need to know directly what it is registered for.

What did we do?

We set the menu up to ask the user intelligently whether they want to use history left over from a previous attempt to play the game that was interrupted. Then, we set the game up to load the saved user input stream. The game should now be fully functional!

Game over – wrapping it up

You can now install the game on your device and experiment with things like force-stopping it in the middle of a game or powering the device off and on again. When you relaunch the game app and reselect the level, it should give you the choice of picking right up where you left off!



Can you take the HEAT? The Hotshot Challenge

One of the things the game scene does that is left unused in this pass through the project is creating walking sprites for the main character. See if you can modify the world object to make the character face in the right direction whenever he moves or attempts to move!

Project 3

TranslationBuddy – Fast App Development for any Field

While Corona is probably best known as a tool for building games and e-books, it includes a few goodies that make it considerably easier to launch nearly any application, including the remote-site frontend apps so popular with business clients. Its high-level architecture allows many of these apps to be generated with a fraction of the code that would be required to create them in Java or Objective-C. As a simple example of how we can leverage this, we'll build an application that collects text entry from the user and submits it to the Microsoft Translator API.

What do we build?

TranslationBuddy is a small, multi-screen app. It will allow the user to enter a new phrase in English, or switch to a list of phrases they have translated before. Once they have entered a new phrase or selected an old one, it displays the whole phrase, along with its Japanese translation. A **Back** button allows the user to back out of the results into whichever screen they loaded it from. A tab bar at the bottom also allows the user fast access to the entry and history screens.

Although the app in its presented form provides only one specific conversion, we'll make sure to write the translation module so that it doesn't internally make assumptions about which source or target language will be chosen.

What does it do?

For this project we are going to be building a fast, simple app that requires minimal graphical content. It showcases Corona's versatility by providing a useful, readily-adaptable application without a lot of coding from scratch or downloading third-party libraries that might be expensive, awkward to learn, or both.

1. We'll start by creating a general-purpose translation module that logs into the Microsoft Translator API. This is a web service that you register for through the Windows Azure marketplace; you can send text to it in various languages to receive translations back in various languages. This will give us practice at making Internet requests to remote services.
2. We'll create a total of three scenes: one to collect requests from the user; one to present phrases which were previously translated; and one to display the translation of a phrase. Because Microsoft Translator is a metered service with limited access, we'll re-use previously collected translations instead of requesting new ones.
3. We'll also create a tab bar that floats above the rest of the app, allowing the user to quickly switch between the two phrase selection scenes.

Why is it great?

This project will provide some experience at using tools that Corona provides to simplify common tasks. Some have been visited already, such as using Corona's JSON library to quickly store and retrieve complex data objects in files or data streams or using the `storyboard` library to facilitate switching between different data views, but the biggest aid to quickly launching apps with familiar user interfaces is the `widget` library.

The `widget` library creates visual objects that fulfill common interface tasks, such as push buttons, scrollable lists, tab bars, and draggable sliders. These objects are created with simple default settings that allow them to be deployed very quickly in prototypes, but also allow complex custom settings that allow their appearances to be tailored to designer specifications with considerable detail. And since their core behavior is event-based, this detail can be added without redesigning the functional part of the code.

In cases where large numbers of custom objects are needed, Corona even allows you to create a theme module, a Lua file that explains all your desired customizations for the widgets you'll use, or even use multiple themes, such as in cases where you might want a menu used to give commands to a game separated from a menu that chooses settings for that game.

The other feature we're going to leverage is Corona's `network.request` API. This reduces the complex two-way communications and synchronizations used to transfer data back and forth from the Internet to a single call with a few options. In the simplest cases where you need a file off the Internet, you can use the even simpler `network.download` function.

Because network requests can take time, Corona expects you to give it back control so that it can continue to run other things while waiting for your response. We'll take a look at a way to organize the code so that it acts more like a single function that can be paused until the response comes in, by using a Lua feature called a **coroutine**. This is a powerful and versatile feature that we'll use more as time goes by.

How are we going to do it?

This project will be shorter and simpler compared to the previous one. The design work requires little analysis and there's no iteration over the various modules. However, we also need to prepare an account so that we can connect to the web service that will do the actual translation. In this project we will be covering the following points:

- Summarizing the design
- Creating the account
- Assembling the translator
- Displaying results
- Soliciting input
- Maintaining a history

What do I need to get started?

You'll need the `presentation` folder of the `Project 3` file pack. This folder contains images for the tabs in the navigation bar and a mask that will be used to prevent visual effects from bleeding over each other during transitions from one view to another.

You'll also need a Windows Live ID. If you don't already have one, you can obtain one for free from the Microsoft Sign up page at `https://signup.live.com/signup.aspx`.

Summarizing the design

This is just as important for any other piece of software as it is for games. A clear understanding of the software's goals and the different interactions that will take place between the user and software will be a vast help in keeping the software usable. This project is the last one in which we'll explicitly cover the design that states what the software needs to do.

Getting on with it

Because this app has only a few features, the design will stay fairly simple.

1. Start by explaining the overall function:

   TranslationBuddy will provide translations of short phrases and words from English into Japanese, and preserve a history of translations that have been previously obtained so that they can be viewed again.

2. Specify the user experience along the principal interaction path (how the user is expected to use the software most often):

   When the app opens, an editable text field is displayed above a tab bar with two options: **Translate** *and* **History**. **Translate** *is the current option. When the text field is selected, the device's native text entry facility appears; when it is dismissed, the view pans to the right-hand side to reveal a list of the entered text and the translation below it. The tabs remain visible so that the user can return to enter or select another translation easily. The result display is also topped with a bar that includes a* **Back** *button to return to whatever selection method opened the results screen.*

3. Explain the user experience along any other interaction paths (there is only one in this case):

   The History tab button reveals a table of previous English sentences that have been entered. This list can be scrolled up and down to reveal parts that don't fit on the screen. Tapping a sentence will reveal the translation of that sentence much as if it had been just entered in the Translate view.

4. Finally, explain any other non-functional requirements; that is, details that might be possible to change without completely upsetting the chosen solution:

   Translations will be obtained from the Microsoft Translator API. Because this service has limited access, the app will store received translations with the sentences that produced them to avoid needlessly repeating requests.

What did we do?

We've outlined what tasks the app will perform and summarized the interface elements that will implement those tasks. We've specified a couple of what project managers call *non-functional requirements*, specifications about how things will be done, and the parameters that operations will be performed within, such as which engine will provide the translations and that it must conserve the service subscription. These requirements could be changed without stopping the app from doing its job, but they might have other side-effects, such as damaging the app's business model, if it had one.

Creating the account

In order to obtain translations via the Web, we need to make requests of an appropriate service. I selected Microsoft Translator for this project because it allows you to set up a free subscription to translate up to 2 million characters of text a month; using Google Translate instead would be similar, aside from the URLs used to communicate, but Google Translate does not, at the time of this writing, have any option for free or trial use through third-party programs.

The app assumes that all users of the app will use the same subscription information, so all translations done with TranslationBuddy will come out of the same 2,000,000 character per month service allowance. If you were launching this app as a product for wide use, you would presumably need to subscribe to a higher-capacity service plan and have a revenue model in place to support it, but for educational purposes, this basic account should be more than adequate.

Getting ready

Have a Windows Live ID created and available, either the one you already had or the one you create just for this project. They are freely available from Microsoft.

Open Corona and create a new scene-based project from the Corona splash screen as you did with the first two projects, called `TranslationBuddy`. Create a new file called `credentials.lua` in the project folder, open it, and enter the skeleton table that will hold the authorization information you're going to create:

```
return {
  id = [[]],
  secret = [[]],
}
```

Getting on with it

Creating a Microsoft Translator API application account is done through the Azure services marketplace:

1. Start by directing your web browser to `https://datamarket.azure.com/dataset/bing/microsofttranslator` and let the page load. Price listings should appear on the right-hand side of the page; scroll down past the alarmingly large numbers at the beginning and locate the bottom entry in the list, which should say **SIGN UP** rather than **BUY**; click on **SIGN UP**. You may be required to sign in with your Windows Live ID at this point.

2. If you have not used your Live ID to obtain services through the Azure Marketplace before, you may need to enter some information about yourself in order to proceed. Once you have completed the form fields and selected **CONTINUE**, you should be presented with the **Terms of Use** for the Microsoft Windows Azure Marketplace. Feel free to read them over; if you cannot comply with them, move on to the next project. Check **I accept the Terms of Use** and click on **REGISTER** when you are ready to proceed.
3. At this point, you are nearly done signing up for the Translator service access. Check **I have read and agree to the above publisher's Offer Terms and Privacy Policy** and click on **SIGN UP**. This should take you to a purchase receipt page.
4. Now that you have an allowance for the service, you need to create a secret key and client ID that the app will use to identify itself to the service in order to have its requests accepted. Go to `https://datamarket.azure.com/developer/applications/` in your browser and find the section towards the bottom of the page entitled **Registered Applications**. Unless you have used this Live ID for other development work, it will probably say "You do not have any registered applications." Click on the **REGISTER** button.
5. A form appears requesting identifying information about your application. For the client ID, enter `CoronaHotshotProject3XXX`, replacing `XXX` with your own initials; this must be unique across all applications registered on the marketplace, so it may be necessary to add your birth date or something else to the client ID to make it unique.
6. For the name, enter `TranslationBuddy`. The client secret field should be pre-populated; leave it alone. A redirect URL is required, but will not be used for this service; you can use a common URL such as `https://example.com`.
7. Enter `Mobile translation app` for the description, and click on **CREATE**. You should return to the **Developers** page, but your new name should now be visible under **Registered Applications**.
8. Follow the **Edit** link at the right-hand side of the screen next to it. This reopens the form you used to register the application. We'll enter the required information into the `credentials.lua` file so that the application can load it as needed.
9. The client ID is permanent and can't be selected, so on the line in `credentials.lua` that reads `id = [[]]`, you'll need to type in the same client ID in between the pairs of brackets as follows:

```
return {
  id = [[CoronaHotshotProject3XXX]],
  secret = [[]],
}
```

10. The secret is much more complicated and easy to mistype, but you can also select and copy the contents of the text box and paste them into the brackets (be careful to select all the characters in the text box, and nothing else on the page):

```
return {
  id = [[CoronaHotshotProject3XXX]],
  secret = [[a0B9c8D7e6+F5g4H3i2J1/]], -- not a real secret; paste
in your own!
}
```

The double brackets indicate a Lua long string literal. The contents are converted into a string just as if you had used single quotes or double quotes, but nothing inside is a special character; everything is stored into the string contents exactly as you type it into the source. Long string literals can even span multiple lines and will include the new line characters as part of their content. They're good to use for strings that might have special character content, like XML markup. See section 2.1 of the Lua 5.1 manual for details.

Save and close `credentials.lua`. You can save the details for the application on the developer site, and log out of Live if you're so inclined.

What did we do?

We registered an application with a shared communication secret that will allow us to authenticate our app as a legitimate user of the service. We stored the authentication information where we can easily load it into our app's Lua environment to incorporate it into our network requests. We'll be able to use this information in our app to obtain something like a cookie that we can include in our translation requests to show the service that they're genuine.

What else do I need to know?

Treat your application as secret and client ID as confidential information! If they become public, the odds are good that some stranger in another country will start using them to freeload off your Azure subscription and use up your service allowance for himself/herself.

Assembling the translator

It's generally better to assume that you'll want to reuse your code later for something else, so avoid building it around one specific use. Our translator module will consist of factory functions for different translation services (although we're only going to include one) that produce functions that perform actual translation work. Any translator function produced by a factory function should take the same arguments, but the various factories might have different arguments to represent different authentication criteria that the different services might use. We only need one factory for Microsoft Translator, which will need to know the client ID and shared secret in order to create a translator function that can authorize itself.

Getting ready

Create a new file in the `TranslationBuddy` project directory, `translation.lua`, and open it in the text editor of your choice.

Getting on with it

First, we'll lay out the skeleton of the module as we've described it, and then we'll start fleshing out the factory for Microsoft Translator-enabled translation handlers.

```
local translation = {}

function translation.microsoft(client_id, client_secret)
   return function(text, from, to, completion)
   end
end

return translation
```

Here we establish two things: what information is needed to create a translator that can draw on a specified Microsoft Translator subscription (the client ID and secret that we created and stored in `credentials.lua`), and what information a translator engine needs in order to do a translation: the text to translate, the language that text is in now, the language you want it translated into, and a callback function. Because fulfilling a translation request may require network traffic (and will, in the case of our Microsoft module), it can't return the result as soon as the function is called. Instead, the translator receives a function as an argument that it will call with the finished translation once it's done. This is similar to the way you make a `network.request` call; you provide it with a function that will process the result once you get it back, and then go on with whatever else you're doing.

Gatekeeping requests

We've already observed that your program might be able to take actions while it's waiting for the translation results, so we'll need to keep track of whether the translator engine is ready to receive requests.

```
function translation.microsoft(client_id, client_secret)
  local ready
  return function(text, from, to, completion)
    if ready then
      -- initiate translation
      return true
    else
      return false
    end
  end
end
```

By returning `true` or `false` according to whether or not the engine was ready, programs using the engine to request translations can know whether they can expect their requests to be processed or not.

Consuming requests

What we're creating is a pattern that computer scientists sometimes call a **consumer**, something that you keep supplying requests to and it keeps processing them. We'll create a function that loops infinitely, requesting new text to be translated, processing the translations, and passing them off to the callbacks supplied with them. Then, we'll integrate it with the pieces that will allow it to go into hibernation while it's waiting for a new translation request, or for a network response to arrive:

```
function translation.microsoft(client_id, client_secret)
  local ready, authorization
  local function translate(source)
    while true do
      local text, from, to, completion = source()
      local translation = fetch_translation(text, from, to,
authorization)
      completion{
        name = 'Translation', action = 'complete';
        fromLanguage = from, toLanguage = to,
        source = text, result = translation
      }
    end
  end
  return function(text, from, to, completion)
    if ready then
```

Here's the basic form of our consumer function. It loops infinitely, collecting a translation request from a function provided to it when it starts. It calls another function to process that request using an authorization it maintains using the credentials it was created with (the arguments to the `translation.microsoft` function). Finally, it constructs an event structure describing the results of the request and hands it off to the function that was supplied with the request. Then it lets the loop repeat.

Maintaining authorization

Note the addition of the authorization variable. Microsoft Translator requires each request to be accompanied by an access token. The access token is requested from an authorization server, and is good for a certain amount of time (typically 10 minutes) from its issuance. The client ID and shared secret must be supplied when the access token is requested for the token request to be granted. So we need to make sure before any request that we have an authorization, and that it's up to date:

```
        local text, from, to, completion = source()
        if not authentication or authentication.expiration <= os.time()
then
          authentication = obtain_token(client_id, client_secret)
        end
        local translation = fetch_translation(text, from, to,
authentication)
```

Linking the translation loop

The way the function is designed right now, it tries to run continuously, which isn't going to work. It has to give control back to Corona whenever it shifts back into waiting mode, or the responses to its requests will never be delivered. The way to do that is by running it inside a Lua coroutine.

Coroutines are a little bit like threads in other multitasking environments (and in fact calling the type function on a coroutine object returns the string thread). They allow a function to be run semi-separately from the rest of the program, so that it can bookmark itself and go into a wait mode; when the main program calls it again, it picks up from the point where it bookmarked itself, as if nothing had happened.

To allow coroutines (and the main program) to communicate with each other, they can also pass values back and forth as arguments and returns when they hand off control. The first time a coroutine is started after being created, the body function can receive arguments through the start process and use them normally. The translate loop function takes one argument, a function that it can call to obtain the details of its next translation request.

Once it's ready to wait for the first request, the loop will yield control of the program back to the main thread that started it by calling a dedicated function, `coroutine.yield`. Once the main program resumes the coroutine again, by passing it a translation request, the `coroutine.yield` function will return the values that were passed into the restart process by calling `coroutine.resume`. Let's start by seeing how this function works:

```
local ready, authentication
local function submissions()
  ready = true
  local text, from, to, completion = coroutine.yield()
  ready = false
  return text, from, to, completion
end
local function translate(source)
```

This function releases control back to the main program and waits for the program to pass it the details of a translation request. So before it does that, it marks the `lock` variable's `ready` as `true` so that the function that receives the requests will know not to refuse them. Once it's been restarted, it marks the translator as busy again so that if any duplicate requests are posted, they won't interfere with its progress.

The translator factory now needs to set up its coroutine and link the supply function to it:

```
      source = text, result = translation
    }
  end
end
local self = coroutine.create(translate)
coroutine.resume(self, submissions)
return function (text, from, to, completion)
  if ready then
```

Notice that after creating the coroutine, we start it and pass it the `submissions` function we created to collect requests. Since the first thing it does is call that function, the coroutine yields almost immediately, after setting `ready` to `true`. So the first time the control function (the function returned at the end of the factory) is called, it will resume the function from where it is paused, waiting to assign the particulars of the translation request.

This means we're ready to enable passing those requests into the coroutine when it's ready and the calling code has a translation request:

```
coroutine.resume(self, submissions)
return function (text, from, to, completion)
  if ready then
    coroutine.resume(self, text, from, to, completion)
```

```
        return true
      else
        return false
      end
    end
  end
```

Since we supply the various parameters as extra arguments to `coroutine.resume`, and as the coroutine was already yielded, those parameters will be the return values from the `coroutine.yield` call.

Handling the network requests

Every translation request requires an HTTP `GET` call to a URL on a Microsoft Translator server. This call needs to be accompanied by an access token that was issued in the last 10 minutes; if no such access token exists, a new one needs to be obtained. We call functions for both of these operations in the code we've already created, but they don't exist yet. The trick is that each function has to yield and set itself up to be resumed in order to return.

1. We'll start with the one that gets called on every translation request:

```
local translation = {}

local request_address = [[http://api.microsofttranslator.com/v2/
Http.svc/Translate?text=%s&from=%s&to=%s]]

local function fetch_translation(text, from, to, authentication)
end

function translation.microsoft(client_id, client_secret)
```

2. The `request_address` property is something of a constant. It describes the Internet address, as defined by the instructions for communicating with the Microsoft Translator engine, to which translation requests need to be sent. Notice the `%s` tokens embedded in it; these can be replaced, by using Lua's `string.format` function, with the values of the appropriate arguments.

```
local function fetch_translation(text, from, to, authentication)
  local self = coroutine.running()
end
```

3. This records the currently running coroutine. We'll need this information to make sure that we can resume the right one once the response to our request is received.

```
local function fetch_translation(text, from, to, authentication)
  local self = coroutine.running()
  local function extract(event)
  end
```

```
  network.request(request_address:format(url.escape(text), from,
to), "GET", extract, {headers = {Authorization = "Bearer " ..
authentication.access_token}})
end
```

4. If you're not familiar with how web server communications work, each request or response placed via the Hypertext Transfer Protocol contains a body and a number of headers, although either can be empty. Each header has a name and a value. The Microsoft Translator API specifies that each request needs a header called **Authorization** whose value is the word `Bearer` followed by a space and the value of the access token.

   The `extract` function, which we'll fill out in a moment, is supplied to `network.request` so that it has a way of notifying our program once the request is complete.

The function `url.escape` converts arbitrary text into a form suitable for inclusion in URLs by replacing illegal characters with escaped equivalents. It's part of the `LuaSocket` library, created to add low-level networking capability to Lua, and included in Corona.

```
  network.request(request_address:format(url.escape(text), from,
to), "GET", extract, {headers = {Authorization = "Bearer " ..
authentication.access_token}})
  return coroutine.yield()
end
```

5. Having filed our network request, we sit back and wait for the results to be returned to us. But in order for those results to make it to the right place, we have to make sure that the response function sends them there.

```
local function extract(event)
  local content = event.response:match("%b<>([^<>]+)%b<>")
  coroutine.resume(self, content)
end
```

6. The response comes back as a very simple XML document, one that just has a single tag. While we could write or use a library to parse XML into Lua data structures, such an operation is serious overkill for a result that will consistently be only a single tag and its text contents. The `%b<>` search expression in Lua pattern matching searches for a less-than sign, a greater-than sign, and everything in between, allowing us to quickly and simply eliminate the tags and just capture the text content.

   By passing the string along to `coroutine.resume`, we allow the function that yielded to pass it back as a normal return.

Renewing the access token

The process of requesting a fresh access token is actually very similar. The network address is different, the request is a POST rather than a GET, and the client credentials are stored in the request body rather than the URL, but the form, of posting a request, yielding, and then allowing the response to resume the thread is very similar:

```
local translation = {}

local token_provider = [[https://datamarket.accesscontrol.windows.net/
v2/OAuth2-13]]
local token_request = [[grant_type=client_credentials&client_
id=%s&client_secret=%s&scope=http://api.microsofttranslator.com]]

local function obtain_token(client_id, client_secret)
  local self = coroutine.running()
  local function update(event)
  end
  network.request(token_provider, "POST", update, {body = token_
request:format(url.escape(client_id), url.escape(client_secret))})
  return coroutine.yield()
end

local request_address = [[http://api.microsofttranslator.com/v2/Http.
svc/Translate?text=%s&from=%s&to=%s]]
```

The authorization information is actually returned as a JSON description of a record. Corona includes a function to build a Lua table equivalent to a JSON-encoded value, so converting it back will be trivial. The authentication response has four values, but we're only actually interested in two: the actual access token, and the duration it's good for (this is typically 10 minutes, but there's no reason to rely on that). Since it doesn't store a record of when the token was issued, we'll need to store that, too.

1. We convert the encoded response into a table with separate fields:

```
local function update(event)
  local results = json.decode(event.response)
end
```

2. In order to preserve the time the token was issued, and more importantly, when it expires, we collect the current time, broken down into a table with separate fields for the different components: hours, minutes, and for our purposes, seconds, which is what the token's duration is provided in. So now we're going to adjust that time:

```
local function update(event)
  local results = json.decode(event.response)
  local now = os.date('*t')
end
```

3. We make two modifications to the time we've gotten. We add the duration of the token in seconds to get the expiration time. We also clear any specification about whether the time provided is in Daylight Savings Time or not. This is only rarely a problem; it only makes a difference if the current time and the specified time are on opposite sides of Daylight Savings Time starting or ending. But when it does occur, it leads to bugs that are extremely frustrating and hard to pinpoint, such as causing you to label a token that lasts 10 minutes as not expiring for an hour and 10 minutes:

```
    local now = os.date('*t')
    now.sec, now.isdst = now.sec + results.expires_in, nil
  end
```

Make a habit of clearing DST data from time specifications whenever you use a time and date table to increase, or decrease, a duration or point in time by a certain amount. I spent multiple days isolating a bug in a Lua program that was caused by automatic DST adjustment.

4. Record the expiration time into the authorization object so that we can track how much time it has left:

```
    now.sec, now.isdst = now.sec + results.expires_in, nil
    results.expiration = os.time(now)
  end
```

Feeding the table back into `os.time` converts it into a pure numerical measure of time that can be compared as being less (earlier) or more (later) than another time.

5. Finally, return the usable authorization object back into the thread that's waiting on it:

```
    results.expiration = os.time(now)
    coroutine.resume(self, results)
  end
```

The translation module should now be ready for action!

What did we do?

We constructed a blueprint for creating functions that accept translation requests, and implemented that blueprint for the specific service that was specified in our design. We used a coroutine to allow the function flow to act like the outside of a program rather than the inside, making the program flow more natural and easier to write. We also successfully engineered a two-way communication between our program and a remote server, submitting authentication credentials and storing a short-term authorization to complete requests.

What else do I need to know?

If you want to test this part, try commenting out the contents of `main.lua` and adding the code:

```
local translation = require "translation"
local credentials = require "credentials"
local translator = translation.microsoft(credentials.id, credentials.
secret)
local function report(event)
  print(event.result)
end
translator("Translation Buddy", "en", "ja", report)
```

The Corona Terminal should, after a moment, print out 翻訳の相棒. Testing code modules and individual functions is an important way to make sure you catch errors early, before you've written a lot of code that's dependent on something that may be broken.

Make sure you remove this code and uncomment the previous contents before proceeding.

Displaying results

Once we obtain the translation results, we need to display them. The design says that we can have a separate view for this, which slides in to display the entry view. A scene seems like a good fit for each of these views; we'll create one that stores the various text elements of the translation in the rows of a table view.

Getting ready

Copy the `scenetemplate.lua` file in the project directory to a new file named `result.lua` and open it.

Getting on with it

The design says that this display will contain a list of the text and its translation, and a bar with a button to return to the previous view. We want the button bar to appear above the table, but simply creating a table view and placing a rectangle above it runs into a snag; while Corona's `tableView` objects offer a lot of convenience, they don't currently have an easy way to clear or reset them. The simplest way to make sure a table is up to date with its intended contents is to create the table each time the scene is visited, and delete it when the scene is no longer visible; this means that we can always enter the contents into a fresh, empty table.

To keep elements stacked correctly as they're destroyed and recreated, what we'll do is create a couple of groups and establish them as zones into which different sorts of content can be stacked, so that we have a place into which we can put new tables so that they will always be behind the controls.

This is a common pattern in graphics programming, to create layers within a stack of visual elements, so that regardless of the order of items in each layer, they're always separated from the layers above and below them. The zones this creates typically are reserved for elements with specific meanings or purposes and are sometimes referred to as **strata**, a word taken from geology and referring to a stack of distinct layers with different characteristics.

Constructing the strata

Find the `createScene` function for this scene, and replace the placeholder comment with the creation of some groups that will serve as our strata, or layers. Groups follow the same rules we've just described; if group 2 is in front of group 1, then each thing in group 2 is in front of all things in group 1:

```
function scene:createScene( event )
  local group = self.view

  self.Data = display.newGroup()
  group:insert(self.Data)
  self.Controls = display.newGroup()
  group:insert(self.Controls)
end
```

The control layer doesn't need to be refreshed from one trip through the scene to another, so we can create it now. In the style of the iOS navigation bars, we'll create a blue field to block out the background and give the button a natural place to live. Because this is not an immersive app like a game, we'll leave room for the status bar at the top of the screen.

```
  group:insert(self.Controls)
  self.Controls.y = display.statusBarHeight
  display.newRect(self.Controls, 0, 0, display.contentWidth, display.
contentHeight * 0.08)
    :setFillColor(0x90, 0xA0, 0xC0)
end
```

Adding the controls

Buttons and other widgets offer a lot of customization options, but most of these have sensible defaults. To facilitate providing only the information you need to, each widget constructor takes a table as its single argument, which can contain only the settings that you want to override or specify. The default button appearance is a white rounded rectangle with a black border and a black label, but we want to adjust our button to make it look more at home on our blue background.

```
    :setFillColor(0x90, 0xA0, 0xC0)
  local buttonHeight = self.Controls.height * 0.8
  self.Back = widget.newButton {
    left = display.contentWidth * 0.05, top = buttonHeight * 0.1;
    label = "Back", yOffset = buttonHeight * -0.1, labelColor =
{default = {0xFF, 0xFF, 0xFF}, over = {0x70, 0x70, 0x70}};
    height = buttonHeight, width = buttonHeight * 2.75;
    fontSize = buttonHeight * 0.66;
    onEvent = relay;
  }
  self.Back:setFillColor(0xB0, 0xC0, 0xF0)
  self.Controls:insert(self.Back)
end
```

We set the button's size to be a little shorter than the bar it's on, its font size to be small enough to fit in that height, and its width to be in proportion.

Buttons can have many more customizations, such as using embossed text for the label, or using custom images instead of simple rounded rectangles. The most important customization a button has, however, is its behavior. Buttons are supposed to do something when you activate them, so the creation of a button also takes a function that specifies what will happen when the user interacts with the button by touching the button, releasing a touch that began in it, or sliding their finger around after touching the button.

Widgets handle events a little strangely. Rather than dispatching events directly to the widget as an event target, they take a function at creation time which is the only receiver of that widget's events. However, that function can also take charge of broadcasting events onto the widget, allowing them to be handled the way most other events are.

```
local scene = storyboard.newScene()

local widget = require "widget"

local function relay(event, ...)
    event.target:dispatchEvent{name = 'buttonEvent'; target = event.
target; phase = event.phase}
```

```
  return true
end

-- Called when the scene's view does not exist:
function scene:createScene( event )
```

You can still register your event response function directly to your button if you prefer; this makes the code a little less flexible, but sometimes code needs efficiency more than it needs flexibility. Buttons also allow you to specify up to two different listeners to handle press and release events separately from each other; if you use any of these, you should not specify the general `onEvent` function.

Once our button receives events the way we expect it to, we can set the scene up as a listener, to respond to the button's activation by returning to the previous scene. The name we specified for button events is, fittingly, `buttonEvent`:

```
  self.Controls:insert(self.Back)
  self.Back:addEventListener('buttonEvent', self)
end
```

Since the scene is being used as the listener, it needs a method named after the event to handle it:

```
local function relay(event, ...)
  event.target:dispatchEvent(event, ...)
  return true
end

function scene:buttonEvent(event)
  if event.phase == 'ended' then
    storyboard.gotoScene(self.Origin)
  end
end

-- Called when the scene's view does not exist:
function scene:createScene( event )
```

Note that we return to the scene with its name stored in `scene.Origin`. For this to work properly, we'll need to set that value whenever the scene is started, which we'll set up after one important step.

Preloading scene visuals

Unlike our previous scenes, this scene will be moved onto the screen using a transition. This means that parts of the scene will be visible before the scene is completely on the screen; the `enterScene` event fires for a scene once its entrance transition is complete and it's fully on the screen. If we wait until this point to go through the usual process of filling in the scene's contents, it will appear to slide in mostly empty and then suddenly fill in with info once it stops moving. Since most people will find this a jarring experience, we want to populate the scene before it starts to appear. We'll do this by changing the data population from taking place in the `enterScene` event to the `willEnterScene` event, which fires as soon as the scene change is requested. So change the `scene:enterScene` function to `scene:willEnterScene`, and change the event registration from `scene:addEventListener('enterScene', scene)` to `scene:addEventListener('willEnterScene', scene)`. Also change `scene:exitScene` to `scene:didExitScene` and update that listener registration in the same way.

Now we'll take care of that stray variable that wasn't assigned yet.

```
function scene:willEnterScene( event )
  local group = self.view

  self.Origin = storyboard.getPrevious()
end
```

Creating the list display

As said when we created the view strata, the easiest way to make sure the list is updated whenever the scene is reloaded is to destroy the table and create it from scratch:

```
function scene:willEnterScene( event )
  local group = self.view

  self.Origin = storyboard.getPrevious()

  local output = widget.newTableView{
    id = "results_output";
    width = display.contentWidth, height = display.contentHeight;
    topPadding = self.Controls.contentBounds.yMax, bottomPadding = 50;
  }
  self.Data:insert(output)
end
```

While table views support an enormous list of customizable options, the focus in this project is using the ready-to-go nature of widgets to make a project run very easily and quickly, so we're using defaults when reasonable. The only thing we're specifying is the padding on the top and bottom. These make sure that no part of the data being displayed is trapped under the control bars at the top or bottom. We set the top padding to start at the bottom of the control section, which allows for both its height and the height of the status bar.

Adding rows to the display

Now we need to add each piece of text to the table. As the project is laid out right now, only two items will ever be displayed: the English source text and its Japanese translation. However, it's not inconceivable that we might do something in the future like have a piece of text entry translated into several languages simultaneously. We'll start by adding a row to the table view for each item being displayed:

```
    self.Origin = storyboard.getPrevious()

    local texts = event.params
    local output = widget.newTableView{
      id = "results_output";
      width = display.contentWidth, height = display.contentHeight;
      topPadding = self.Controls.contentBounds.yMax, bottomPadding = 50;
    }
    self.Data:insert(output)
    for i, text in ipairs(texts) do
      output:insertRow{
        -- construct rows
      }
    end
end
```

Now we run into a small stumbling block. There are still a small number of useful facilities that Corona doesn't provide direct support for, and one of these is measuring the space required to display a piece of text. Fortunately, most of these missing facilities have some way to work around them, and we can determine how many lines are needed to show a given string by setting a hidden text object to show the string and checking how tall it becomes:

```
    self.Origin = storyboard.getPrevious()

    local width, height = display.contentWidth, display.contentHeight
    local texts = event.params
    local output = widget.newTableView{
      id = "results_output";
      width = display.contentWidth, height = display.contentHeight;
```

```
    topPadding = self.Controls.contentBounds.yMax, bottomPadding = 50;
  }
  self.Data:insert(output)
  local measure = display.newText(group, "", width, height, width, 0,
native.systemFont, height * 0.075)
  measure.isVisible = false
  for i, text in ipairs(texts) do
    measure.text = text
    output:insertRow{
      rowHeight = measure.height;
    }
  end
end
```

We just need to make sure that we clean up this temporary text object once we're done with it:

```
    output:insertRow{
      height = measure.height,
      onRender = displayText,
    }
  end
  measure:removeSelf()
end
```

Finally, for each row that we provide, we need to explain to Corona how the row will appear and what data it will contain. You can use different functions for different rows in cases where they don't all appear the same, but all we're doing is showing the text of each item. The default color of new text objects is white, which doesn't show up well (or in fact at all) on a white background, so we set each new text object to display in black:

```
  local texts = event.params
  local function displayText(event)
    display.newText(event.row, texts[event.row.index], 12, height *
1/256, width, 0, native.systemFont, height * 0.075)
      :setTextColor(0x00)
  end
  local output = widget.newTableView{
    id = "results_output";
    width = display.contentWidth, height = display.contentHeight;
    topPadding = self.Controls.contentBounds.yMax, bottomPadding = 50;
    onRowRender = displayText;
  }
  self.Data:insert(output)
```

Cleaning up the scene after each use

Since we're creating a new table view each time this scene is visited, we need to make sure they get cleaned up, or we'll come back to overlapping displays (and creeping memory leaks):

```
function scene:didExitScene(event)
  while self.Data.numChildren > 0 do
    self.Data[1]:removeSelf()
  end
end
```

What did we do?

We developed a complete life cycle for a scene that will probably be reloaded multiple times while the app is running, working around some limitations of our target platform (which you'll find you need to do in nearly any programming environment). We separated visual environments logically as well as visually by storing them in separate groups, and used built-in capabilities provided by Corona to display buttons and complex program output to the user with a minimum of work on our part.

Soliciting input

Of course, a translation engine isn't much use unless you can enter text that you want translated. To collect this, we'll use a native display object, a text box. The user can enter a large string of text into this box, and the program will be notified when the user closes it.

Getting ready

Copy `scenetemplate.lua` into a new file called `entry.lua` and open it.

Getting on with it

Native display objects aren't quite the same as widgets, although they often look similar. A widget is actually a Corona display group containing other rectangles, text, and other Corona display objects, created and managed with internal Lua code. This means that you can put widgets into other display groups and control their front-to-back ordering. Native objects, on the other hand, are created by the device's operating system at Corona's request. Although they have many of the same properties as display objects, such as x and y position and visibility, they don't move with their parent group or inherit any characteristics like alpha, and they always appear in front of any standard Corona display objects you create, regardless of group; that means they aren't affected by things like storyboard transitions.

Creating a backdrop

We'll give the scene a blank white background to help the elements stay visually separate:

```
function scene:createScene( event )
  display.newRect(self.view, 0, 0, display.contentWidth, display.
contentHeight)
end
```

Creating the text box

Because text boxes and other native interface elements aren't really part of the Corona canvas, they won't move or hide when you change scenes with the storyboard. You can deal with this in a few ways, but we're going to settle it by deleting the text box when the scene is closed, and recreating it when the scene is re-opened:

```
function scene:enterScene( event )
  local group = self.view

  local width, height = display.contentWidth, display.contentHeight
  self.Entry = native.newTextField(width * 0.075, height * 0.1, width
* 0.85, height * 0.66)
end

-- Called when scene is about to move offscreen:
function scene:exitScene( event )
  self.Entry:removeSelf()
  self.Entry = nil
end
```

Processing the user input

Native text objects work more closely with the event model, dispatching events onto the text object. These events have the name `userInput` and update their listeners about several sorts of events, such as when the editor or keyboard appears, when the text has changed, when the user has hit *Enter*, or when the keyboard has been closed, which is usually interpreted as a cue to process whatever the box contains. Listeners can use these triggers to perform complex tasks like auto-fill, but in our case, we're only interested in submission.

1. We'll start by registering the scene object as a listener on the text box:

```
  self.Entry = native.newTextField(width * 0.075, height * 0.1,
width * 0.85, height * 0.66)
  self.Entry:addEventListener('userInput', self)
end
```

2. We'll add a function to the scene object to process the events sent to its listeners. We're interested in two specific event actions, `submitted`, which indicates that the user has hit a button like done or *Enter*, and `ended`, which indicates that the text box is no longer receiving input; either the keyboard has been hidden or a different text field is now in charge of input:

```
local scene = storyboard.newScene()

function scene:userInput(event)
  if event.phase == 'submitted' then
    native.setKeyboardFocus(nil)
  elseif event.phase == 'ended' then
  end
end

-- Called when the scene's view does not exist:
function scene:createScene( event )
```

3. When the box is submitted, we'll just remove focus from it, hiding the keyboard and triggering the other event, where we'll do the actual work:

```
function scene:userInput(event)
  if event.phase == 'submitted' then
    native.setKeyboardFocus(nil)
  elseif event.phase == 'ended' then
    local originalText = event.target.text
    -- process translation
  end
end
```

4. When the input is complete, we'll want to process it. To do that, we need a translator engine that we can make the request of. To create one of those, we need authorization info to contact the translation service. Move up to the top of the file, after the storyboard calls, and add two new required calls:

```
local scene = storyboard.newScene()

local credentials = require "credentials"
local translation = require "translation"

function scene:userInput(event)
```

5. Pass the credentials to the factory to create a translator:

```
local credentials = require "credentials"
local translation = require "translation"

local translator = translation.microsoft(credentials.id,
credentials.secret)

function scene:userInput(event)
```

6. Now, step back to the `scene:userInput` function. Remember that the translator returns `true` to indicate that it's started processing your request, or `false` to show that it was busy and couldn't take your request at this time. We'll use this to provide user feedback:

```
elseif event.phase == 'ended' then
  local originalText = event.target.text
  if translator(originalText, 'en', 'ja', output) then
    native.setActivityIndicator(true)
  end
end
```

7. The activity indicator is a system-dependent signal that tells the user that the system is busy and their actions can't be processed at the moment. While it's visible, Corona ignores user events like touches. On iOS, it looks like a spinning spokes pattern that fades its way around the circle. Also notice that we call the translator with an argument called `output`. This needs to be the callback function that will process the translated text when the translation is finished. Frame it in before the `scene:userInput` function:

```
local translator = translation.microsoft(credentials.id,
credentials.secret)

local function output(event)
end

function scene:userInput(event)
```

8. The first thing it needs to do is dismiss the activity indicator, to tell the user that the translation is done and to tell the system that user input can be accepted again:

```
local function output(event)
  native.setActivityIndicator(false)
end
```

9. The result display screen expects an array of strings, so it creates one:

```
local function output(event)
  native.setActivityIndicator(false)
  local texts = {event.source, event.result}
  displayResults(texts)
end
```

10. Finally, it calls a function to pass this array to the result screen. This function is pretty simple and we probably wouldn't make a separate function for it except that we'll need it again in the next section:

```
local translator = translation.microsoft(credentials.id,
credentials.secret)

local function displayResults(texts)
  storyboard.gotoScene( "result", {effect = 'slideLeft'; params =
texts })
end

local function output(event)
```

Tying the pieces together

Save `entry.lua` and open `main.lua`. It should still contain the template text that gets added when Corona creates a new project for you. Let's start by setting the status bar to the light appearance typical of productivity apps:

```
display.setStatusBar(display.DefaultStatusBar)

local storyboard = require "storyboard"
storyboard.gotoScene( "scenetemplate" )
```

Interestingly, `display.DefaultStatusBar` is frequently not the status bar that apps display by default. Many devices default to showing `display.TranslucentStatusBar`.

Now, make sure that the app starts by displaying the input screen:

```
local storyboard = require "storyboard"
storyboard.gotoScene( "entry" )
```

At this point, you should be able to test the app in the simulator or on your device. We have only one part left to make sure that the app is fully functional and efficient.

What did we do?

We created a view that uses built-in operating facilities to easily collect a string of text from the user. We created a translation engine that authenticates with our app's unique passcodes and used it to obtain translated text. We used a familiar, well-understood interface element to tell the user to wait until we received the translation back and forwarded it to our display view.

Maintaining a history

To help the user with common phrases and save bandwidth and translation allowance, the app will store a history of requested translations and their results. We'll give the user the option to view them, and use them to avoid duplicate network requests if the user enters text that has already been translated.

Getting ready

We'll store the history file in the app's `Documents` folder, which sync software generally backs up to the user's computer. Each line will be a JSON-encoded copy of the arrays of strings that holds the original text and the translation.

Open the `main.lua` file in the `TranslationBuddy` folder, if you don't already have it open from the previous task.

Getting on with it

The history view and the entry view both need access to the history file, so it's important for them to agree on its location. Lua doesn't have symbolic constants, but we can create a global variable and never change it; both modules can then share it as a common file path.

1. Near the top of `main.lua`, before the storyboard calls, add a line to define that location, in the directory that gets backed up by sync software:

```
display.setStatusBar(display.DefaultStatusBar)

PATH = system.pathForFile("translation.history", system.
DocumentsDirectory)

local storyboard = require "storyboard"
```

2. Touch the file path to make sure that the file exists. We don't want to use the file at this time, so we close it as soon as we open it:

```
PATH = system.pathForFile("translation.history", system.
DocumentsDirectory)
io.open(PATH, 'a')
  :close()

local storyboard = require "storyboard"
```

Opening a file in append mode (that's what the `'a'` indicates) is the safest way to guarantee that the file exists, because it will create an empty file if there wasn't one, but it will not overwrite or erase the contents of an existing file, which write mode (indicated with a `'w'`) will.

3. Now we can leave `main.lua` for the moment and modify `entry.lua` to use this file as a cache. Find the `scene:userInput` function; if we already have the translation saved, there's no need to send it to the translation engine. We'll look at each line, and see if its array starts with the entered text:

```
local originalText = event.target.text
for line in io.lines(PATH) do
  local history = json.decode(line)
  if history[1] == originalText then
    displayResults(history)
    return
  end
end
if translator(originalText, 'en', 'ja', output) then
```

   If we find such an array, which indicates that the text was translated before, we'll simply proceed to the results screen using that saved text, and bail out of the function without calling the translator.

4. For this to work, we also need to save anything we do get from the translator and add it to the file, so we'll have it next time. We'll do this in the `output` function that handles translation results:

```
  storyboard.gotoScene( "result", {effect = 'slideLeft'; params =
texts })
end

local function output(event)
  native.setActivityIndicator(false)
  local texts = {event.source, event.result}
  save(texts, PATH)
  displayResults(texts)
end
```

5. Since the translator is never even called unless the requested text was not in the file, we can simply append it to the end, along with its translation:

```
local translator = translation.microsoft(credentials.id,
credentials.secret)

local function save(entry, path)
  local log = io.open(path, 'a')
  log:write(json.encode(entry), '\n')
  log:close()
end

local function displayResults(texts)
```

Viewing the history

At this point, the entry module is silently caching translation results in order to save service costs. However, the design also calls for the app to let the user view this history and review previous translations. This calls for another scene. Save `entry.lua`, and copy `scenetemplate.lua` into a new scene file, `history.lua`. Open this file in your preferred editor.

The history view will use a table view to display the English lines that were sent to the translator. Like the result module, it will create the table fresh each time the scene is launched, but it's even simpler; it needs no extra controls, so no stratum is required:

```
function scene:enterScene( event )
  local group = self.view

  self.History = widget.newTableView{
    id = "translation_history";
    width = display.contentWidth, height = display.contentHeight;
    topPadding = display.statusBarHeight, bottomPadding = 50;
  }
  group:insert(self.History)
end

-- Called when scene is about to move offscreen:
function scene:didExitScene( event )
  self.History:removeSelf()
end
```

Similarly to the result module, the history will use an array table to store all the translation requests to display or execute. However, it will use single-line text objects to display just the first part of the English submissions, and each entry will be based on one line in the history file:

```
    local group = self.view

    self.Data = {}
    local function displayHistory(event)
      display.newText(event.view, self.Data[event.row.index][1], 4,
event.row.height * 0.125, native.systemFont, event.row.height * 0.75)
        :setTextColor(0x00)
    end
    self.History = widget.newTableView{
      id = "translation_history";
      width = display.contentWidth, height = display.contentHeight;
      topPadding = display.statusBarHeight, bottomPadding = 50;
      onRowRender = displayHistory;
    }
```

The table will fill in this array with lines from a text file:

```
    group:insert(self.History)
    for line in io.lines(PATH) do
      table.insert(self.Data, (json.decode(line)))
      self.History:insertRow{
        height = display.contentHeight * 1/12;
      }
    end
end
```

Unlike the result view, rows in this table should be selectable to load their contents in the result view. Table views make this easy by taking an `onRowTouch` handler that processes touch events on the rows and notifies them when they've been pressed, released, or swiped. We're only interested in releases in this case.

The function is basically a copy of the `displayResults` utility function in `entry.lua`:

```
        :setTextColor(0x00)
    end
    local function loadHistory(event)
      if event.phase == 'release' then
        storyboard.gotoScene("result", {effect = 'slideLeft'; params =
self.Data[event.row.index]})
      end
    end
```

```
self.History = widget.newTableView{
  id = "translation_history";
  width = display.contentWidth, height = display.contentHeight;
  topPadding = display.statusBarHeight, bottomPadding = 50;
  onRowRender = displayHistory;
  onRowTouch = loadHistory;
}
```

Enabling both views

We want users to be able to request translations through either the new entry or the history view; there's an established convention for this in mobile apps, and Corona provides support for it through the **tab bar** widget. A tab bar usually takes up the bottom of the screen and has a few icons, often with text labels, that can be tapped to switch between different pages of an app's interface. Like the other widgets, the tab bar is highly customizable, but its default appearance is frequently quite adequate, so the only things you have to specify are where to put it and what buttons to put on it.

1. First, reopen `main.lua` and load the `widget` module:

```
io.open(PATH, 'a')
  :close()

local widget = require "widget"

local storyboard = require "storyboard"
```

2. Visual elements that are loaded after the storyboard is initialized will float over the shared layer that storyboard scenes are loaded into, so we'll create the widget at the bottom of the file:

```
storyboard.gotoScene( "entry" )

widget.newTabBar{
  top = display.contentHeight - 50;
  buttons = {
    {
      id = "entry";
      label = "Translate";
      width = 32, height = 32;
      defaultFile = "presentation/translate-up.png", overFile =
"presentation/translate-down.png";
      onPress = pickScene;
      selected = true;
    },
  }
}
```

The `id` field for a button can be any Lua value you care to use, and I use the names of the scenes they will load to keep the code simple. The `label` field is a string that will be displayed under the icon for the tab. It can be blank, but leaving icons unlabeled makes them much less useful; very few are as immediately intuitive as their creators believe. The `width` and `height` fields are mandatory in current versions of the widget API, and just specify the dimensions of the button icon. The `defaultFile` and `overFile` fields specify the icons to be used for the tab when it is selected (over) or deselected (default). The selected value should only be set on one button, and indicates that that button should use its up image when the bar is loaded. The `onPress` field should be a function that handles presses on the tab; it can be shared between different buttons if they have some way for the function to distinguish them (like the `id` value).

3. The second button will be very similar:

```
  buttons = {
    {
      id = "entry";
      label = "Translate";
      up = "presentation/translate-up.png", down = "presentation/
translate-down.png";
      onPress = pickScene;
      selected = true;
    },
    {
      id = "history";
      label = "History";
      width = 32, height = 32;
      defaultFile = "presentation/history-down.png", overFile =
"presentation/history-up.png";
      onPress = pickScene;
    },
  }
```

4. Finally, they just need the function that will launch their appropriate scenes. Fill it in just before the widget is created:

```
storyboard.gotoScene( "entry" )

local function pickScene(event)
       storyboard.gotoScene(event.target._id)
end

widget.newTabBar{
```

For reasons that aren't clear, current versions of Corona store the IDs assigned to tab bar buttons in the `_id` field rather than `id` as previous versions did. This might change in a future version of Corona.

Now you can load the app and switch between the two views. However, unless your history only consists of very small phrases, you may notice an odd glitch when you select a sentence from the history.

Keeping the effects clean

To keep the table short, the history view displays only the beginning of each English sentence in a single-line text object. However, the rest of each text is still hanging off the right edge of the screen. This isn't an issue until the scene slides left to make room for the result display, dragging the extra text across the view. We'll fix this with a mask.

The file `presentation/masking-frame.png` is pretty simple. Since the screen size is specified in the `config.lua` file as being 320 x 480, the mask file consists of a white area (which will reveal everything in its target object normally) surrounded by a 4 pixel black border on all sides. Black prevents anything in the target object from showing up at that spot. So, using this mask crops out everything around the edges of the target screen. We'll load the mask with the history module, and attach the mask to the view whenever it's created:

```
local json = require "json"

local frame = graphics.newMask("presentation/masking-frame.png")

-- Called when the scene's view does not exist:
function scene:createScene( event )
  local group = self.view

  group:setMask(frame)
end
```

The mask can be positioned on its target using the `maskX` and `maskY` values, which specify where to put the center of the mask compared to the masked object's local origin. This is usually the center of the object, but for groups, it's whatever point is defined as the group's point (0, 0). That's frequently the top-left corner, depending on where you placed the group's children inside it. So, we'll move the mask to line up with the center of the screen:

```
function scene:createScene( event )
  local group = self.view

  group:setMask(frame)
  group.maskX, group.maskY = display.contentCenterX, display.
contentCenterY
end
```

If you try it again, you should not see any more spill-over from the history view into the results as they slide in!

What did we do?

We built a scene that uses a slightly more complex table view. We populated it with data from a file rather than a table, and made it respond to touch actions to select the specified entries. We also created another commonly-used, familiar user interface object to switch between the two related tasks the app can perform; viewing old translations and viewing new translations.

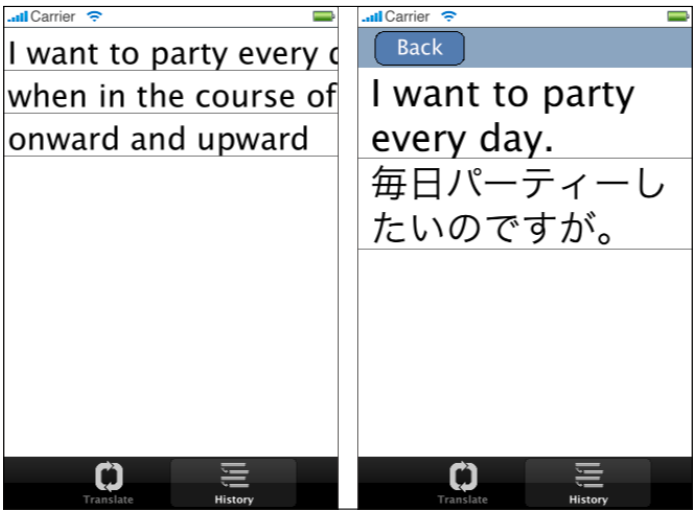
What else do I need to know?

When creating tab bar icons, it's common to leave any part that might be considered the background of the icon, transparent so that it can be superimposed over any bar or background. It's also typical to make some distinctive difference between the selected and unselected forms of an icon, such as grayscale versus color, or flat versus embossed.

To work properly with the graphics rendering engine, images used as masks must have a height and width that are multiples of four, and they should have a border all the way around the edges of black pixels, at least 3 pixels thick. Grays can also be used to make the masked target partially transparent. We'll explore masks in detail in *Project 6*, *Predation*

Game over – wrapping it up

At this point, you've learned how to make Corona log on to a remote service and process the results, how to make an extended task pause and resume itself with new input from the main process, and how to produce a familiar, basic user interface with very little work. The ability to produce straightforward, frontend apps like this can be very useful as a freelancer, making it easy to take requests from business clients, who often pay better than game designers. The following screenshot shows an example of our app:



Can you take the HEAT? The Hotshot Challenge

Depending on whether you're more practical or entertainment-minded, we have two recommendations for the Hotshot Challenge on this project. We made sure that the result display module isn't limited to two strings, so both challenges center around making use of that.

For one option, have the entry module obtain translations of the text in several different languages, rather than just one, and present them all to the user. If you're feeling ambitious, put the flag for each language's country next to its text in the results view!

If you're feeling a little more whimsical, develop the app to work more like the site `translationparty.com`; have it translate the sentence into another language, then translate the result back into the original language, and see how much it's changed. Do this a few times back and forth and see what it turns into!

Project 4

Deep Black – Processing Mobile Device Input

Games developed for traditional platforms have used a variety of input schemes to let the player control the game. These have included joysticks, buttons, keyboards, and pointing devices such as mice. On tablets, smartphones, and other mobile devices, many of these input forms are missing or awkward; keyboards are on-screen representations with small buttons, or extra devices that have to be carried separately and physical buttons are few. However, most of these mobile devices also have some new inputs such as motion and tilt sensors. Designing games to make the most of these capabilities also requires you to understand how to read them, which is easy, and how to interpret that input, which is a little harder.

What do we build?

Deep Black will be a game in the style of Asteroids, where you have to dodge flying space rocks while using your ship's laser to whittle them down to size. Instead of using left turn, right turn, and thrust buttons, the player will control the ship's direction and motion by tilting the device and by shaking it to use their hyperspace jump. Any touch on the screen will fire the ship's weapon as long as it is being held down.

What does it do?

The game will model the player's spaceship and asteroids flying around. It will allow the player to launch laser shots, which will break up the rocks when they collide, accumulating points for the player. The game will also wrap the player's ship and bullets, as well as the asteroids, around to the other side of the screen as they move over the edge. The game will control the player's motion according to the angle at which the device is being held. The degree to which the tilt is, in line with the front-back axis of the player's ship will accelerate it forward or backward; the degree to which the tilt is perpendicular to the ship's angle will turn the ship to one side or the other. This means that the input will dynamically change meaning as the ship adjusts to it.

This will also be our first game project to use lives for the player to determine game progress.

Why is it great?

This will be our first project that uses Corona's physics engine. This will save us most of the trouble of controlling objects' speed, moving them around the screen, and detecting when they bump into each other. We'll also see how to use the power of the physics library to choose which objects can or can't bump into each other; for instance, in asteroids-style games, the asteroids don't smash when they pass over each other.

Corona's physics engine uses Erin Catto's popular Box2D free collision modeling library, and sometimes you can combine info from the Box2D online manual (found at `http://www.box2d.org/manual.html`) with Corona's physics documentation to produce more detailed answers to your tough physics questions.

We'll discuss how to collect tilt input from the device's accelerometers, and how to use it to determine the angles at which the device is being held. We'll compare these angles to the angle at which the player's ship is positioned in order to separate it into turn and thrust controls.

How are we going to do it?

For this project, we'll review the design, rather than creating it. While an indie programmer will often be designing their own games and then coding from their designs, any programmer at a studio is likely to be coding from a design document given to them by a designer or design team.

- Creating the player and receiving events
- Processing raw physical events
- Bridging physical events to game events

- Creating the world rules
- Creating enemies and controlling collisions
- Responding to fire controls and creating bullets
- Responding to collisions and handling lives
- Recognizing kills and recording score

What do I need to get started?

First, read over the `design.txt` file and consider how to solve some of the problems presented there:

- Lots of things need to move continually, some under constant speed (rocks and bullets), others under adjustable acceleration (the player ship). Moreover, we need to recognize when these things come into contact, so that bullets can split or destroy rocks and rocks can destroy the spaceship. The physics module addresses all of these needs.
- The game needs to recognize seven different inputs from the player in four categories: turn left, turn right, accelerate forward, accelerate backward, start firing, stop firing, and jump through hyperspace. Five of these inputs will come from the accelerometer: four from tilt, and one from shaking. The other two will come from the touch screen.
- An interface layer will be needed to display the current score and available lives. Lives are easy to display with a series of repeated images, but the score display is more naturally handled as numeric text.

Creating the player and receiving events

The first thing to do is to create the player ship object, both visually and with a physics presence. That will require turning physics on in the scene so that the player's ship can be simulated. Once the object appears, we'll specify how it will respond to certain inputs from the player, in abstract terms; events which specifically describe the desired actions. This will get us ready to supply those events based on physical events produced by the user's handling of the device.

Getting ready

Create a new folder called `Deep Black`, and copy the contents of the `version 0` folder from the `Project 4` project pack directory: the `build.settings` and `config.lua` files that go with any typical Corona project; the `visuals.lua` and `explosion.lua` library files that we used in *Project 1*, *Bat Swat – an Introduction to App Event Cycles*; the `sprite`, `background`, `effect`, and `image` folders and the art files in them; the `main.lua` file; and the `splash.lua` and `game.lua` files. This version of `game.lua` is a bare scene file, a copy of the `scenetemplate.lua` file provided by Corona in new scene-based projects. The step of copying this file into new scenes should be familiar from previous projects.

The `build.settings` file included in this folder is set to launch our game in landscape orientation.

Getting on with it

We'll start by creating an almost blank scene that contains just the player. Open the `game.lua` file and find the `createScene` event handler.

Creating a placeholder world

We'll start by creating a plain colored background. We'll use a color that matches the overall look of the background image we'll use later:

```
function scene:createScene( event )
  local group = self.view

  display.newRect(group, 0, 0, display.contentWidth, display.
contentHeight)
    :setFillColor(0x5E, 0x3F, 0x6B)
end
```

We'll create a single group to hold any objects that are meaningful in the world (later, we'll replace this with a full-fledged `World` object):

```
    :setFillColor(0x5E, 0x3F, 0x6B)
  self.World = display.newGroup()
  group:insert(self.World)
end
```

Because an asteroids game needs a certain amount of space, our sprites are a bit big compared to the screen. As a quick fix, we'll make the world twice as big by scaling it down:

```
  group:insert(self.World)
  self.World.xScale, self.World.yScale = 0.5, 0.5
end
```

For release, you'll typically want to fix up your sprites for the target resolution; they'll look crisper. But when you're in early development, short-term solutions like this one can keep you moving and get you on to the next step right away.

Preparing the physics simulation

We'll be using physics to move ships and bullets around, so go to the beginning of the file and perform the following steps:

1. Load the required module:

```
local storyboard = require( "storyboard" )
local scene = storyboard.newScene()

local physics = require "physics"
```

2. The setup of a new game will be done in the `willEnterScene` event for the game scene; this event is the last warning we get before the scene starts to become visible:

```
  self.World.xScale, self.World.yScale = 0.5, 0.5
end

function scene:willEnterScene( event )
end

function scene:enterScene( event )
```

3. Because the new scene has no guarantees about anything that might have been using the physics environment before, the first thing it will do on starting is wipe any previous physics simulation:

```
function scene:willEnterScene( event )
  physics.stop()
end
```

The first time you start the game scene, an error prints out in the terminal that physics couldn't be stopped because it wasn't started yet. You can safely ignore this message; it won't be visible to your users or affect the play experience.

4. Once we know that the old physics environment, if any, has been cleared, we can set up a new one so that we can use our sprites as physics objects. We'll pause it right away so that the simulation won't run physics during any opening transition:

```
function scene:willEnterScene( event )
  physics.stop()
  physics.start()
  physics.pause()
end
```

5. Since this is a top-down game with no gravity, we configure that fact on the physics environment before proceeding:

```
  physics.start()
  physics.pause()
  physics.setGravity(0, 0)
end
```

6. Finally, once the scene is fully loaded, we'll start the physics simulation in response to the scene's `enterScene` event:

```
function scene:enterScene( event )
  physics.start()
end
```

Setting up the ship object

First, we'll specify the sprite and physics information for the player's ship. Create a new file in your project folder, `spaceship.lua`, and open it for editing.

1. This module will be responsible for adding physics to the ship sprite, so we'll require that module:

```
local physics = require "physics"
```

2. The ship will be a sprite with four different views contained in one image sheet. We'll start by specifying the basic form of the image sheet:

```
local physics = require "physics"

local sheet = graphics.newImageSheet("sprite/player.png",
  {
    sheetContentWidth = 80, sheetContentHeight = 98;
  }
)
```

Unlike our previous image sheets, this one is not uniform in its layout, so we specify the positions of the four frames individually:

```
local sheet = graphics.newImageSheet("sprite/player.png",
  {
    frames = {
      {
        x = 41, y = 1;
        width = 38, height = 50;
      },
      {
        x = 1, y = 1;
        width = 38, height = 50;
      },
      {
        x = 1, y = 52;
        width = 39, height = 45;
      },
      {
        x = 40, y = 52;
        width = 39, height = 45;
      },
    };
    sheetContentWidth = 80, sheetContentHeight = 98;
```

3. To make a sprite from the sheet, we need a sequence definition, which explains which frames are used for which purposes. In this case, the sprite is not actually animated, so each sequence is only one frame long:

```
    sheetContentWidth = 80, sheetContentHeight = 98;
  }
)

local sequences = {
  {
    name = 'level';
    start = 1, count = 1;
  };
  {
    name = 'left';
    start = 4, count = 1;
  };
  {
    name = 'right';
    start = 3, count = 1;
```

```
  };
  {
    name = 'damaged';
    start = 2, count = 1;
  };
}
```

4. This lets us fill in the basic function that will generate a sprite from these definitions:

```
    start = 2, count = 1;
  };
}

return function(parent)
  local self = display.newSprite(parent, sheet, sequences)
  return self
end
```

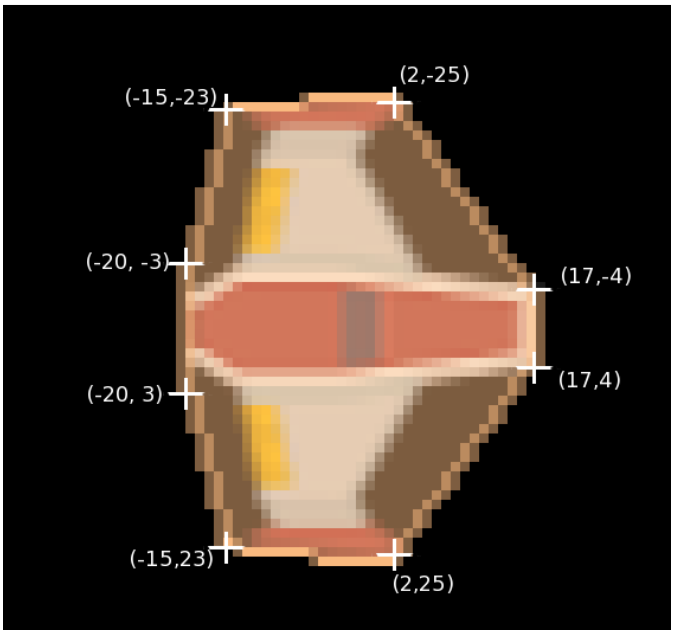
The default shape for a new physics body on an object is a rectangle with the dimensions of the object's bounding box; using this would lead to frustrated players when rocks appear to collide with the empty space around the ship's corners and destroy it. We'll define an octagonal shape that comes closer to the ship's visible shape.

```
  };
}

local body = {
  shape = {-20, -3; -15, -23; 2, -25; 17, -4; 17, 4; 2, 25; -15,
23; -20, 3};
}

return function(parent)
```

You can define arbitrary shapes as an array of values, alternating between x and y values. There are two rules; the points should be defined in clockwise order (given that Corona's coordinate system is upside-down from Box2D's, which specifies counter-clockwise points), and the shape should be convex, with no places where a point is dimpled in between two of its neighbors.



5. We'll specify this shape definition for the body when we add it to the sprite:

```
return function(parent)
  local self = display.newSprite(parent, sheet, sequences)
  physics.addBody(self, "dynamic", body)
  return self
end
```

Attaching behaviors to the player ship

Now that we have the basic physical and visual definition of a ship, we need to define how it responds to certain game events. To keep our code modular and easy to modify later, we'll do this in a separate file. Save `spaceship.lua`, create a new empty file `player.lua`, and open it.

1. The player constructor function will take arguments indicating the event source that the player will listen to for input events, as well as the world group that the player object will belong to, and where to place the player ship in that world:

```
return function(input, world, x, y)
end
```

2. It will construct a spaceship object in that world, and move it to the indicated point:

```
return function(input, world, x, y)
  local self = require "spaceship" (world)
  self.x, self.y = x, y
  return self
end
```

3. Because the sprite image is pointed right, but we want it pointed up, we'll rotate it to face that direction. Setting `isFixedRotation` may seem counterintuitive for a ship that will spin around, but this only specifies to the physics engine that it is not allowed to turn the object when we accelerate it:

```
  self.x, self.y = x, y
  self.rotation = -90
  self.isFixedRotation = true
  return self
```

In games where sprites rotate freely, it's often helpful to create the image files so that the sprites face right by default. This means that we can easily use the rotation angle of the player sprite to calculate things like which direction a bullet fired by the player should travel. If the sprite image faced a different way, we would have to correct the angles each time we made a calculation.

4. We'll register the player to receive commands and other events that it needs from the input object, as well as detect time passing in the world:

```
  self.isFixedRotation = true
  world:addEventListener('clock', self)
  input:addEventListener('Thrust', self)
  input:addEventListener('Yaw', self)
  input:addEventListener('Fire', self)
  input:addEventListener('Jump', self)
  return self
```

5. Whenever time passes in the game (clock events), we'll apply any specified thrust force to the player ship, and rotate it by any input amount. We'll preset these values to `0`:

```
  self.rotation = -90
  self.Burn, self.Turn = 0.0, 0.0
  function self:clock(event)
    if self.Burn ~= 0 then
      local angle = math.rad(self.rotation)
      self:applyForce(self.Burn * math.cos(angle), self.Burn *
math.sin(angle), self.x, self.y)
```

```
    end
    self:rotate(self.Turn * event.delta / 1000)
  end
  input:addEventListener('clock', self)
```

6. To keep gameplay sane, the design specifies a top speed for the player ship, so we'll enforce that:

```
  self.isFixedRotation = true
  self.TopSpeed = 400
  self.Burn, self.Turn = 0.0, 0.0
  function self:clock(event)
    local dX, dY = self:getLinearVelocity()
    if math.pythagorean(dX, dY) > self.TopSpeed then
      local angle = math.atan2(dY, dX)
      self:setLinearVelocity(self.TopSpeed * math.cos(angle),
self.TopSpeed * math.sin(angle))
    end
    if self.Burn ~= 0 then
```

7. When the input source provides new input values for the thrust on the ship or turning angle, based on the tilt of the device, we have to capture those values so that they can be applied by the clock processor:

```
  input:addEventListener('clock', self)
  function self:Thrust(event)
    self.Burn = event.value
  end
  input:addEventListener('Thrust', self)
  function self:Yaw(event)
    self.Turn = event.value
  end
  input:addEventListener('Yaw', self)
```

8. And finally, we also need to set the player sprite to correctly display its *tilted* animations for left and right turns:

```
  function self:Yaw(event)
    local direction = 'level'
    if event.value < 0 then
      direction = 'left'
    elseif event.value > 0 then
      direction = 'right'
    end
    self:setSequence(direction)
    self.Turn = event.value
  end
```

Adding a library function

The `pythagorean` function, which calculates total distance between two points based on their x and y distances, isn't part of the `math` library as pre-loaded, so we'll require it at the start of the player module.

```
require "math.pythagorean"

   return function(input, world, x, y)
     local self = require "spaceship" (world)
```

Now we just need to provide that module. Lua automatically assumes that prefixes in package names separated by periods correspond to folder names in your project, so after you save `player.lua`, create a folder `math` in your project directory and make a new empty file in it called `pythagorean.lua`. Open this file and frame in the addition of a new function to the `math` module:

```
module 'math'

function pythagorean(a, b)
end
```

This function is fairly simple; it applies the $a^2 + b^2 = c^2$ logic of the Pythagorean theorem to find the total distance between two points:

```
function pythagorean(a, b)
  return sqrt(a * a + b * b)
end
```

Given how small this function is, why is it in a module by itself? Mostly because it's a simple, abstract function which isn't naturally a part of one of the other core elements of the game. Several of the game elements can use it, so redefining it for each one doesn't make sense. While it could simply be a module of its own at the top level of the project, it's clearly math-related, so it makes sense to add it to the standard module.

What did we do?

At this point, you should be able to run the code, although all it will do is show you a ship on a purple background. Right now nothing is sending the events that it's waiting to respond to; addressing that will be our next task.

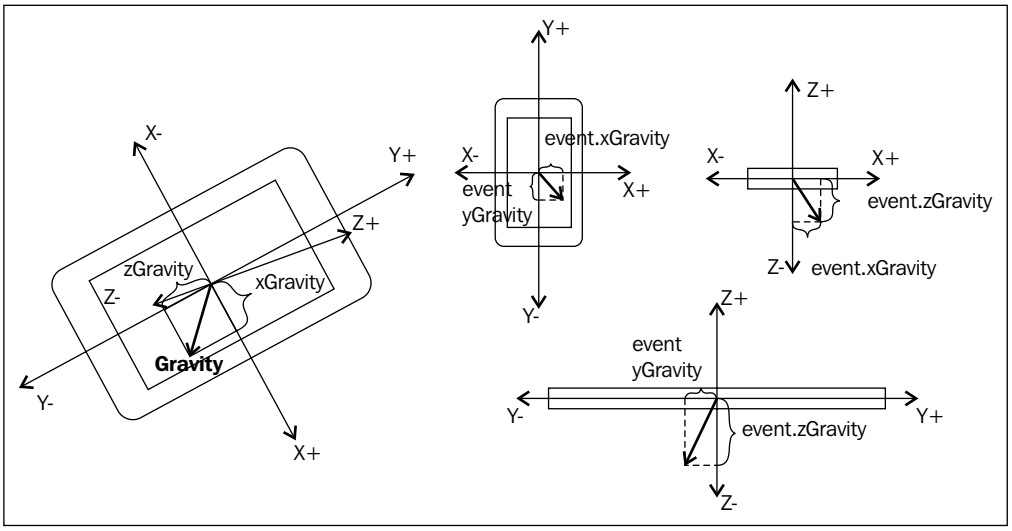
So far we've established the basics of an environment for our player object to occupy, constructed a sprite with a physics identity, and prepped it to receive control events. We've also created a general-purpose helper function that can be reused in other modules or projects.

Processing raw physical events

Accelerometer events are complex, reporting three main pieces of information—tilt vector, motion vector, and whether or not the event has been identified as a shake of the device—contained in seven different fields. However, we're interested in only two and a limited aspect of those at that; shake recognition, and the angle of the device to the horizontal (actually two angles; the side-to-side and front-to-back tilts). So we'll attach a listener that processes these events on `Runtime` and sends simplified events with the information in the form we want to deal with.

Getting ready

A little more explanation is in order for this task. The actual math required to get the desired information (the two angles by which the device is off the horizontal plane) only takes two lines of code, but understanding why this math achieves the desired result is less simple.



Corona represents gravity in three vectors; a direction and magnitude in three dimensions, broken into its three Cartesian coordinates. Imagine an arrow pointing from the middle of your phone or tablet towards the focus of gravity, that is, the center of the Earth.

If you're not on the Earth, the arrow will be pointing somewhere else, and might be longer or shorter. This is probably not a concern for most of our readers.

The coordinates of the end of the arrow are provided in the event as if the device were lying flat at the center of the universe; that is, even though—from the outside—we see the Earth as standing still and the device as turning over, from the accelerometer's viewpoint, the x, y, and z axes are fixed to the device and the arrow pointing out gravity is rotating across them.

While the exact length of the arrow may vary—depending on where you are, what device you're using, and from moment to moment—most of the time, we're only interested in its direction. This can be determined from the proportions of the various vector components, regardless of the total vector length; since they're all perpendicular, applying trigonometry is fairly easy. We can look at any two components as being the opposite and adjacent sides of the vector's angle in that plane. The opposite side from a corner of a right triangle divided by the adjacent side to that corner gives you the tangent of the angle of that corner. So, you can feed the ratio of the `zGravity` (opposite) and `yGravity` (adjacent) components to `math.atan` (or more often, give the two separate numbers to `math.atan2`, which is easier and gives more reliable results) to determine the angle by which the device is tipped front-to-back off the horizontal. We can replace `yGravity` with `xGravity` to determine the sideways tilt.

Getting on with it

Create a new file, `input.lua`, in the top level of your project directory, and open it for editing.

1. Create the skeleton of a new listener function, and register it to respond to `accelerometer` events on the `Runtime` target:

```
local function processor(event)
end

Runtime:addEventListener('accelerometer', processor)
```

2. The code will calculate the angles between the device's z axis and the vertical according to gravity, along the device's x and y axes, and the results will be posted back to `Runtime` in a new event named `Tilt`:

```
local function processor(event)
  local theta = math.atan2(event.yGravity, -event.zGravity)
  local phi = math.atan2(event.xGravity, -event.zGravity)
  Runtime:dispatchEvent{name='Tilt'; lateral = phi, vertical =
theta}
end
```

3. The listener will also post a custom event called `Shake` if the device is being shaken. Exactly what this means is up to the host operating system and hardware.

```
  Runtime:dispatchEvent{name='Tilt'; lateral = phi, vertical =
theta}
  if event.isShake then
```

```
        Runtime:dispatchEvent{name = 'Shake'}
      end
    end
```

4. As a last courtesy, the module will return this new listener if the host code wants to be able to turn it on and off (we won't need to in this project):

```
Runtime:addEventListener('accelerometer', processor)

return processor
```

5. Save `input.lua`. In order to have this listener loaded and ready in the rest of our code, open `main.lua` and load the input module at the top:

```
display.setStatusBar(display.HiddenStatusBar)

require "input"

local storyboard = require "storyboard"
```

What did we do?

We created an adapter layer to present the information provided by Corona in a form more easily usable to us. This very simple code extension is flexible enough to be reused in other projects.

Bridging physical events to game events

We now have physical events being passed in a very understandable format. We have a little bit of work left to do to translate these events (which are still specifically about the physical condition of the device) into commands for the player.

Getting ready

The math in this section is almost as abstract as the previous task, because the design specifies that the ship's acceleration is going to be based on how strongly the device's tilt aligns with the ship's facing direction, and the speed with which it turns will be based on how much the direction of tilt runs across the ship's facing.

Getting on with it

Save any other open files and open `game.lua`.

1. Make sure the `pythagorean` distance module is loaded; we'll need it in this file as well:

```
local scene = storyboard.newScene()

require "math.pythagorean"

local physics = require "physics"
```

2. We'll start by explaining how `Tilt` events will be translated into `Thrust` and `Yaw` events on the game scene, before the event handlers:

```
local physics = require "physics"

function scene:Tilt(event)
end

function scene:createScene( event )
```

3. We'll normalize the angles into range from -1 to +1 by using `math.sin`, then we'll determine the angle and length of the 2D vector they create using familiar methods.

```
function scene:Tilt(event)
  local lateral, vertical = math.sin(event.lateral), math.
sin(event.vertical)
  local theta, r = -math.atan2(lateral, -vertical), math.
pythagorean(lateral, vertical)
end
```

Because we're using `math.sin`, we're also moving the angles onto a quadratic scale as the tilt increases. This can often produce perfectly acceptable results, especially since most people playing tilt-based games don't tilt the device very far.

4. Since we subtract the player's current rotation from the specified angle, we can redefine the lateral and vertical values as being across the direction of facing and along the direction of facing:

```
  local theta, r = -math.atan2(lateral, -vertical), math.
pythagorean(lateral, vertical)
  theta = theta - math.rad(self.Player.rotation)
  lateral, vertical = math.sin(theta) * r, math.cos(theta) * r
end
```

5. The other thing we want to do is clip small values for both inputs, so that the player has some margin where they're not turning or accelerating, even if they can't hold the device exactly flat (which is nearly impossible):

```
local threshold = 1/6

function scene:Tilt(event)
  local lateral, vertical = math.sin(event.lateral), math.
sin(event.vertical)
  local theta, r = -math.atan2(lateral, -vertical), math.
pythagorean(lateral, vertical)
  theta = theta - math.rad(self.Player.rotation)
  lateral, vertical = math.sin(theta) * r, math.cos(theta) * r
  if math.abs(lateral) < threshold then lateral = 0 end
  if math.abs(vertical) < threshold then vertical = 0 end
end
```

6. Once this is done, we can submit the two values for `Thrust` and `Yaw` as events for the player object to catch:

```
  if math.abs(vertical) < threshold then vertical = 0 end
  self:dispatchEvent{name='Thrust'; value = vertical * 1/3}
  self:dispatchEvent{name='Yaw'; value = lateral * 64}
end
```

7. Then all we need to do is to set the scene to receive `Tilt` events whenever the scene is fully loaded, and stop receiving them when the scene leaves the foreground:

```
function scene:enterScene( event )
  physics.start()
  Runtime:addEventListener('Tilt', self)
end

function scene:exitScene( event )
  Runtime:removeEventListener('Tilt', self)
end
```

Tracking time passage

There are two challenges with running game objects directly off of the `enterFrame` event sent to `Runtime`:

- It lists only the time right before each frame, without indicating how much time has actually passed.
- If objects that want to update themselves all register individually for `enterFrame` events of `Runtime`, then if you want to do something like pause the game, you have to find each of these objects individually and disengage them, or build each listener to check some kind of common `isPaused` value.

We'll use a custom event, `clock`, to solve both of these things. The listener for this event will track the time of consecutive `enterFrames` to include the elapsed duration between them. Because any objects that need to know about time passing in the game will listen to the game scene itself for these clock events, the game scene itself can be responsible for turning them off by disconnecting itself from the `Runtime` events that drive them. We can turn only one listener on and off instead of an indefinite number.

The `player` object is already programmed to listen for these events, so we need to create the bridge that will send them.

1. First, we'll specify how the scene will respond to `enterFrame` events when it's listening to them:

```
local physics = require "physics"

local clock, previous = 0, system.getTimer()
function scene:enterFrame(event)
end

local threshold = 1/6
```

2. The listener will compare the time of the current frame to the last time recorded:

```
function scene:enterFrame(event)
  local elapsed = event.time - previous
end
```

3. It will increment the total accumulated time by the amount elapsed, and record the current time as the new previous time:

```
function scene:enterFrame(event)
  local elapsed = event.time - previous
  clock, previous = clock + elapsed, event.time
end
```

4. Next it will post that passage of time to the scene so that any objects that need to can observe it:

```
  clock, previous = clock + elapsed, event.time
  self:dispatchEvent{name = "clock"; clock = self, delta =
elapsed, time = clock}
end
```

5. We'll add a function that starts the clock, saving the current time so that the first frame received has a previous frame time to check against, and register the listener for `enterFrame` events:

```
  self:dispatchEvent{name = "clock"; clock = self, delta =
elapsed, time = clock}
end
```

```
function scene:Start()
  previous = system.getTimer()
  Runtime:addEventListener('enterFrame', self)
end

local threshold = 1/6
```

6. Another function will stop the clock by discharging any time passed since the last update and removing the listener:

```
  Runtime:addEventListener('enterFrame', self)
end

function scene:Stop()
  local elapsed = system.getTimer() - previous
  self:dispatchEvent{name = "clock"; clock = self, delta =
elapsed, time = clock}
  Runtime:removeEventListener('enterFrame', self)
end

local threshold = 1/6
```

7. Because time passes primarily within the world, whenever the scene is running, the game will respond to clock events by reposting them to the world object:

```
  Runtime:removeEventListener('enterFrame', self)
end

scene:addEventListener('clock', scene)
function scene:clock(event)
  if self.view then
    self.World:dispatchEvent(event)
  end
end

local threshold = 1 / 6
```

8. Finally, the scene will call these functions when it begins or ends:

```
function scene:enterScene( event )
  physics.start()
  Runtime:addEventListener('Tilt', self)
  self:Start()
end

-- Called when scene is about to move offscreen:
```

```
function scene:exitScene( event )
  Runtime:removeEventListener('Tilt', self)
  self:Stop()
end
```

What did we do?

This task has been about interpreting the intended meaning of fundamental events; converting the angle of the device into instructions for the player object, and controlling how the passage of time for the device's clock controls the passage of time for the game.

At this point, you should be able to build the project for the device and test how the player ship moves as you tilt the device in various directions.

Creating the world rules

The world of the game serves three practical functions and one cosmetic one. We will use it to provide a starry background, to enhance the ambiance of the game, as well as to provide a reference for the speed and direction of the motions of things in that world. We will enforce one of the rules from the design; that all objects in the world wrap from one edge to the opposite edge as they move. We will also allow the world to generate a randomly chosen point within its own bounds, which we will use for the player's hyperspace faculty. Finally, the world will encapsulate its own method for removing all mobile objects from itself, clearing out everything except the background.

Getting ready

Create a new file at the top level of your project, `world.lua`, but do not open it yet.

Getting on with it

Start by changing the world construction in `game.lua` to call the module you're about to create. The world will need to know what target will supply it with events (it will use `clock` events to continually enforce some of its rules as the physics engine moves things around).

```
  display.newRect(group, 0, 0, display.contentWidth, display.
contentHeight)
    :setFillColor(0x5E, 0x3F, 0x6B)
  self.World = require "world" ()
  group:insert(self.World)
end
```

Save `game.lua`, open your new `world.lua` file, and fill in a function that creates a new group.

```
return function()
  local self = display.newGroup()

  self.xScale, self.yScale = 0.5, 0.5
  return self
end
```

At this point, the code is carrying out the same function that it was doing before we modularized it.

Filling the visible field

To fill the visible area of the world group, we'll use a pair of nested `for` loops, one to fill a single row, the other to fill each row of the screen.

1. First, we'll get the bounds of the screen in the new group's local coordinates, so we know the space we need to cover:

```
return function()
  local self = display.newGroup()

  self.xScale, self.yScale = 0.5, 0.5
  local xMin, yMin = self:contentToLocal(0, 0)
  local xMax, yMax = self:contentToLocal(display.contentWidth,
display.contentHeight)

  return self
end
```

2. Next, we'll iterate over these dimensions, jumping by the declared size of the background each time:

```
  local xMax, yMax = self:contentToLocal(display.contentWidth,
display.contentHeight)

  for x = xMin, xMax, BG_WIDTH do
    for y = yMin, yMax, BG_HEIGHT do
    end
  end

  return self
end
```

3. In order for this to work, we have to specify the size of the background image, which we'll do at the start of the file:

```
local BG_WIDTH, BG_HEIGHT = 256, 256

return function()
  local self = display.newGroup()
```

4. Then, it becomes fairly easy to construct the images:

```
    for y = yMin, yMax, BG_HEIGHT do
      display.newImage(self, "background/starBackground.png", x,
y)
    end
```

5. After the loop, we'll memorize the number of objects used in the group, so that we know which children are the players, rocks, or bullets:

```
      display.newImage(self, "background/starBackground.png", x,
y)
    end
  end
  local firstMob = self.numChildren + 1

  return self
end
```

Managing the world bounds

Now, we'll add two functions to the world object, to wrap objects within it or to select a random point in it.

1. The wrapping will be performed each time the game object updates the clock, which should happen after each time the physics engine moves things around:

```
  self:addEventListener('clock', self)
  function self:clock(event)
  end
  return self
```

2. We need to consider each object in the world that can move, which will all be on top of the tiles we added at the beginning, which threshold we memorized after creating the background:

```
  function self:clock(event)
    local width, height = xMax - xMin, yMax - yMin
    for i=firstMob, self.numChildren do
    end
  end
```

3. We already have the bounds calculated from when the background was filled in, so it's fairly easy to assess each object's correct position after wrapping; the object's current position, modulus of the world's width, plus the world's origin:

```
    for i=firstMob, self.numChildren do
      local child = self[i]
      child.x, child.y = (child.x - xMin) % width + xMin, (child.y
- yMin) % height + yMin
    end
```

Generating random locations

Since we already have the world's bounds, generating random points is fairly easy. We'll attach a function to the world that the game can call on demand.

```
  input:removeEventListener('destroyScene', self)
end

function self:Random()
end
return self
```

Lua's `math.random` format can generate a number in any range of integers if you give it the smallest and largest allowed values. Since we can't be sure that the group's global bounds are perfect integers, we'll adjust them toward the center of the range.

```
function self:Random()
  return math.random(math.ceil(xMin), math.floor(xMax)), math.
random(math.ceil(yMin), math.floor(yMax))
end
```

Clearing movable objects

If a given world can be reused between games, it's important to make sure that objects from a previous game aren't accidentally carried over into a new one. Since the world recalls the point of division between the background objects and the functional objects, this is fairly easy.

1. Attach the function constructor after the `Random` function body:

```
    return math.random(math.ceil(xMin), math.floor(xMax)), math.
random(math.ceil(yMin), math.floor(yMax))
  end

  function self:Clear()
  end
  return self
```

2. Go through all of the world group's children, working backward until you reach the last background object:

```
function self:Clear()
  for i = self.numChildren, firstMob, -1 do
  end
end
```

3. Remove each child that represents a game object:

```
function self:Clear()
  for i = self.numChildren, firstMob, -1 do
    self:remove(i)
  end
end
```

The world group should be ready for use now. We should make one change to the game object to make use of these new features; save `world.lua` and return to `game.lua`.

4. Add a new scene event registration after `exitScene` for the `didExitScene` event:

```
scene:addEventListener( "exitScene", scene )
scene:addEventListener( "didExitScene", scene )

scene:addEventListener( "destroyScene", scene )
```

This event fires to indicate that the scene in question is no longer visible on the display at all, as a result of a scene transition.

5. Add a handler to support these events:

```
  self:Stop()
end

function scene:didExitScene( event )
  self.World:Clear()
end

function scene:destroyScene( event )
```

6. This handler should clear objects from the world, once it is safely invisible to the user, in preparation for reusing the scene.

What did we do?

We automatically generated enough image tiles to cover the screen, set up a rule to keep objects on the screen, and added a way for the game to request a random point as extensibly as possible. Using this structure for the world makes it easier to update if any of the parameters of the game were to change later in development.

No game design is ever final until release at the earliest. The easier you, as the developer, can make it to accommodate change requests, the less time will be consumed in recoding and extra testing.

At this point, you can test it and see the player ship return to the other side of the screen when it slides off one side.

Creating enemies and controlling collisions

Now the game is ready to start adding some challenges. First we'll add the rocks and make them move, then we can manage destroying the player's ship when a collision takes place.

Getting ready

In this task, we'll not only control the new elements' motion with physics, but we'll also start in on controlling and detecting collisions between elements. The player needs to respond when an asteroid runs into it (by being destroyed), but the design says that the rocks don't collide with each other at all, passing right over each other. We can accomplish this with a Box2D collision group; by specifying the same negative integer as the `groupFilter` field of the filter table in each new object's body description, we tell Corona that these objects never collide with each other. They won't interact physically, and any collision listeners they have won't be called when they overlap.

Positive `groupFilter` values are used with collision masks (discussed in the next project) to make objects collide, even when their masks specify that they wouldn't otherwise.

Getting on with it

The first thing we'll do is create a library for creating the three different sizes of asteroid. We have two different images; we'll fake the middle one by scaling the large one down and mirroring the sprite.

1. Create a file, `asteroids.lua`, open it, and frame in the beginnings of a table module:

```
local physics = require "physics"

local asteroids = {}

return asteroids
```

2. We'll sum up the bodies that will be used to represent the physics of the three different asteroids:

```
local asteroids = {}

local shape = {
  large = {-68,-5; -44,-41; 17,-55; 58,-15; 68,33; 14,55; -54,28;
};
  medium = {-46,3; -37,-18; 11,-37; 46,-21; 39,10; 13,37; -29,27;
};
  small = {-22,9; -17,-19; 13,-21; 22,3; -2,21; };
}

return asteroids
```

3. We'll define a function that can be used to add common asteroid behavior to a display object, based on a supplied physics body, but won't fill it in yet:

```
  small = {-22,9; -17,-19; 13,-21; 22,3; -2,21; };
}

local function asteroid(object, shape)
  return object
end

return asteroids
```

4. Then we'll add three functions to the library, which create the images for the different sizes and use the function we just specified to add their behavior:

```
  return object
end
```

```
function asteroids.large(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  return asteroid(self, shape.large)
end

function asteroids.medium(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  self.xScale, self.yScale = 2/3, -2/3
  return asteroid(self, shape.medium)
end

function asteroids.small(parent)
  local self = display.newImage(parent, "sprite/meteorSmall.png")
  return asteroid(self, shape.small)
end

return asteroids
```

5. The module will be complete once we finish that function. Inside, we'll create a table that combines the provided shape data with a collision filter that prevents the rocks from colliding:

```
  small = {-22,9; -17,-19; 13,-21; 22,3; -2,21; };
}

local rockFilter = {groupIndex = -2}
local function asteroid(object, shape)
  local body = {shape = shape, filter = rockFilter}
  return object
end
```

6. Then we'll use that body description to add physics to the provided display object:

```
local function asteroid(object, shape)
  local body = {shape = shape, filter = rockFilter}
  physics.addBody(object, "dynamic", body)
  return object
end
```

7. We'll give the new body a push and a twist so that it tumbles across the screen in a random direction:

```
  physics.addBody(object, "dynamic", body)
  local push, theta = 1, math.random() * 2 * math.pi
  object:applyLinearImpulse(push * math.cos(theta), push * math.
sin(theta), object.x, object.y)
  object:applyAngularImpulse(25)
  return object
```

8. Finally, so that new rocks don't seem to just blink into existence, we'll use a transition to fade them in. Since the rock is created at full visibility, we'll use `transition.from` to fade it in starting from transparency:

```
local rockFilter = {groupIndex = -2}
local fadeIn = {time = 750, alpha = 0}
local function asteroid(object, shape)
  local body = {shape = shape, filter = rockFilter}
  physics.addBody(object, "dynamic", body)
  local push, theta = 1, math.random() * 2 * math.pi
  object:applyLinearImpulse(push * math.cos(theta), push * math.
sin(theta), object.x, object.y)
  object:applyAngularImpulse(25)
  transition.from(object, fadeIn)
  return object
end
```

The asteroid module should be complete now; save it and open `game.lua`.

Spawning enemies at game start

When the game scene loads and is ready to start playing, we'll add rocks to the scene to serve as challenges.

1. In `game.lua`, find the `willEnterScene` function where we get the game ready, and add a loop at the end to create rocks (we'll only make two at the beginning of the game).

```
  self.Player = require "player" (self, self.World, self.
World:contentToLocal(display.contentCenterX, display.
contentCenterY))
  for i=1, 2 do
    local new = asteroids.large(self.World)
  end
end

function scene:enterScene( event )
```

2. We don't want any of the rocks to start right on top of the player, so we'll choose their starting location based on a minimum radius from the player's location, and a maximum radius to prevent them from wrapping too close.

```
  for i=1, 2 do
    local new = asteroids.large(self.World)
    local theta, r = math.random() * 2 * math.pi, math.random(100,
200)
    new.x, new.y = r * math.cos(theta), r * math.sin(theta)
  end
```

3. You should be ready to test the code again, on the simulator or on the device, and see the asteroids drifting across the screen. If you're patient, you can see them pass over each other or run into the player ship, pushing it around the screen.

What did we do?

We created some simple physics bodies and set them to collide with the player, but not with each other. We set up a response to that collision by using a custom event to keep the code clear and set up multiple responses to that event at the appropriate levels.

Responding to fire controls and creating bullets

It's possible to use the game on your device now for some simple gameplay dodging rocks, although there's no progress or way to lose or advance. It's time to start adding more interactivity by accepting shake and touch inputs to control the player ship's laser fire and hyperspace jump.

Getting ready

Since we won't want the ship to be destroyed by its own lasers, we're going to include it in a collision group much as we did with the asteroids. Start by opening `spaceship.lua` and adding a line to that effect in the body definition used by the player constructor. Save that file and create a new file at the top level of the project, `laser.lua`, for the first stages of this task.

Also, we'll set a timer on the laser objects so that they disappear after a moment if they don't hit anything. To do this, we'll need an object that receives `clock` events to track the passage of time.

Getting on with it

This module is a constructor, so we'll create the framework for a function for it to return as follows:

1. The function will need to know what clock will govern the laser's lifetime, how long that lifetime is, what is firing the laser, which direction it will go, and what art to use for it:

```
return function(clock, life, origin, direction, image)
end
```

2. The function will create a new image object using the specified file:

```
return function(clock, life, origin, direction, image)
  local self = display.newImage(origin.parent, image)
  return self
end
```

3. It will rotate the image to face the specified direction, and place it the appropriate distance from the center of the object firing it:

```
return function(clock, life, origin, direction, image)
  local self = display.newImage(origin.parent, image)
  self.rotation = direction
  direction = math.rad(direction)
  local dX, dY = math.cos(direction), math.sin(direction)
  self.x, self.y = origin.x + self.width * dX, origin.y + self.
width * dY
  return self
end
```

4. It will add a physics identity to the image, and set it as a sensor so that it doesn't transfer any momentum to things that it hits:

```
  local dX, dY = math.cos(direction), math.sin(direction)
  self.x, self.y = origin.x + self.width * dX, origin.y + self.
width * dY
  physics.addBody(self, "dynamic", body)
  self.isSensor = true
  return self
```

5. All lasers will belong to the same collision group as the player, so that they don't collide with the player or each other:

```
local body = {filter = {groupIndex = -1}}

return function(clock, life, origin, direction, image)
  local self = display.newImage(origin.parent, image)
```

6. We'll register the laser to receive `clock` events on its governing object and respond to them:

```
  self.isSensor = true
  clock:addEventListener('clock', self)
  function self:clock(event)
  end
  return self
```

7. In that handler, we'll subtract the elapsed time (in milliseconds) from the remaining duration (in seconds), and clear the laser if it's been used up:

```
function self:clock(event)
  life = life - event.delta / 1000
  if life <= 0 then
    event.clock:removeEventListener('clock', self)
    self:removeSelf()
  end
end
```

8. Finally, we'll give the laser its initial motion before returning it:

```
    self:removeSelf()
  end
end
local speed = 400
self:setLinearVelocity(speed * dX, speed * dY)
return self
```

Dispatching fire control events

Save `laser.lua` and open `game.lua`. We'll respond to physical touch events on the device by dispatching semantic touch events to the game object for the player ship to respond to.

1. Above the `createScene` and other scene event handlers, add a handler for `touch` events:

```
  self:dispatchEvent{name='Yaw'; value = lateral * 64}
end

function scene:touch(event)
end

function scene:createScene( event )
```

2. When a touch starts, we notify the game that the player has started firing the ship lasers.

```
function scene:touch(event)
  if event.phase == 'began' then
    self:dispatchEvent{name = 'Fire'; phase = 'began'}
  end
end
```

3. When a touch ends or is cancelled, we'll notify the game that firing can stop:

```
if event.phase == 'began' then
  self:dispatchEvent{name = 'Fire'; phase = 'began'}
elseif event.phase == 'ended' or event.phase == 'cancelled' then
  self:dispatchEvent{name = 'Fire'; phase = 'ended'}
end
```

4. Since we don't care where on the screen these touches are for these events, we'll listen for touches that reach `Runtime`. Since we only want to respond while the scene is fully loaded, we'll start and stop listening in response to `enterScene` and `exitScene` events:

```
  Runtime:addEventListener('Tilt', self)
  Runtime:addEventListener('touch', self)
  self:Start()
end

function scene:exitScene( event )
  Runtime:removeEventListener('Tilt', self)
  Runtime:removeEventListener('touch', self)
  self:Stop()
```

Responding to fire events

Now that fire events are being transmitted, the `player` object needs to respond to them. We'll use a field on the `player` object, `player.FireCounter`, to track whether there's an active touch being used as a `Fire` command. This field will be nil when the player is not firing, and the number of seconds until the ship can fire again when the player is firing.

1. Save `game.lua` and open `player.lua`. After the registration for `Fire` events, add a handler for them, which will toggle the player between firing and non-firing states:

```
input:addEventListener('Fire', self)
function self:Fire(event)
  if event.phase == 'began' then
    self.FireCounter = 0
  elseif event.phase == 'ended' then
    self.FireCounter = nil
  end
end
input:addEventListener('Jump', self)
```

2. Inside the clock handler, we'll count down the remaining fire time (if any), and reset the timer if it gets below zero. Remember that the clock event's duration is in milliseconds, but the fire counter is in seconds.

```
function self:clock(event)
  if self.FireCounter then
    self.FireCounter = self.FireCounter - event.delta / 1000
    if self.FireCounter <= 0 then
      self.FireCounter = self.FireCounter + 0.5
    end
  end
  local dX, dY = self:getLinearVelocity()
```

3. When the timer resets, we'll also create a laser facing in the direction of the player ship:

```
      if self.FireCounter <= 0 then
        self.FireCounter = self.FireCounter + 0.5
        laser(event.clock, 1, self, self.rotation, "effect/
laserGreen.png")
      end
```

4. This also means that we need to require the laser module at the beginning of the player module:

```
require "math.pythagorean"

local laser = require "laser"

return function(input, world, x, y)
```

Teleporting the player

While we're completing the controls the player can respond to, we'll take a moment to add the hyperspace function.

1. In `player.lua`, add a handler function under the listener registration for `Jump` events:

```
input:addEventListener('Jump', self)
function self:Jump(event)
end
return self
```

2. Move the player to the coordinates specified in the event and stop the motion.

```
function self:Jump(event)
  self.x, self.y = event.x, event.y
  self:setLinearVelocity(0, 0)
end
```

3. Save `player.lua` and return to `game.lua` so that we can add a listener for `Shake` events:

```
  self:dispatchEvent{name='Yaw'; value = lateral * 64}
end

function scene:Shake(event)
end

function scene:touch(event)
```

4. The response to these events is simple; we'll request a random location from the `World` object, and send a `Jump` event with those coordinates to move the player there:

```
function scene:Shake(event)
  local x, y = self.World:Random()
  self:dispatchEvent{name = 'Jump'; x = x, y = y}
end
```

5. We'll register and unregister to receive `Shake` events at the same times we do for `Tilt` events, since they're provided by the same source.

```
  Runtime:addEventListener('Tilt', self)
  Runtime:addEventListener('Shake', self)
  Runtime:addEventListener('touch', self)
  self:Start()
end

function scene:exitScene( event )
  Runtime:removeEventListener('Tilt', self)
  Runtime:removeEventListener('Shake', self)
  Runtime:removeEventListener('touch', self)
```

What did we do?

We created a module to manage temporary laser objects and destroy them cleanly after they hit something. We translated two different events about device information into their equivalent events with game significance, and programmed the player object to respond to them with suitable actions.

Responding to collisions and handling lives

Now we have an environment where rocks collide with the player and with lasers, but not with each other. Right now, however, no responses are installed to these collisions, so default behavior is all that happens; the intangible lasers simply travel until they run out of time, and the rocks and player push each other around the screen. We'll allow collisions to destroy objects, and modify game conditions like the player's remaining lives accordingly.

Getting ready

Like `touch` events, `collision` events can be processed either by the objects actually colliding, or by `Runtime`. Also, like `touch` events, it's almost always easier to process the events correctly in listeners on those objects than in listeners on `Runtime`. Unlike `touch` events, however, `collision` events are dispatched to `Runtime` in a notably different form than they are to individual objects; if you adapt code from a `Runtime` listener for use in an object `collision` listener, you'll need to replace the use of the `object1` and `object2` fields of the event with the fields `target` (the body involved in the collision whose listener this is) and `other` (the other body involved in the collision, which also gets its own event call).

Getting on with it

We need to add collision responses to three things in the game; the player, the rocks, and the lasers. The lasers are the easiest, since their response to collisions will be simply to disappear.

Clearing lasers that hit something

Open `laser.lua` and find the point in the constructor function after it adds the physics body to the new laser.

1. We'll assign a common collision handler to each laser, so they go away after colliding with something:

```
      self:removeSelf()
    end
  end
  self:addEventListener('collision', clean)
  local speed = 400
```

2. The only other thing to do is define the collision handler, so that it marks a laser that collides with something as ready for removal:

```
end
local function clean(event)
end
self:addEventListener('collision', clean)
```

3. First, we'll remove the collision listener from the laser so that there won't be any conflicts if it manages to collide with more than one thing at once:

```
local function clean(event)
  event.target:removeEventListener('collision', clean)
end
```

4. Since the laser is already counting down to its own removal, we'll simply set that counter to expire immediately:

```
local function clean(event)
  event.target:removeEventListener('collision', clean)
  life = 0
end
```

The approach we're using here is required because you can't destroy a physics body in response to a collision; it is still being used by the collision resolution process and could still be interacting with other bodies. We have to defer the object's actual removal until the physics engine has finished with its work and returned control to Corona.

Destroying asteroids

When a laser collides with an asteroid, two collision events will actually be produced for the same collision; one for the laser (already dealt with) and one for the asteroid. We'll use that other event to mark the asteroid to be split up or destroyed.

The three sizes of asteroids will respond differently to being hit; the smallest size gets destroyed and larger sizes spawn two of the next smaller size. Since any of these responses require removing and possibly adding physics bodies, they can't be performed in direct response to the collision. We'll create a collection of clean-up functions that we can attach to the appropriate asteroid sizes at creation.

1. Open `asteroid.lua` and add a table to fill with functions after the library table is defined; these cleaner functions will also use the spawn functions. These new functions will be attached to the various asteroids as `clock` listeners when they're ready to be cleaned:

```
local asteroids = {}
```

```
local resolve = {}
function resolve.large(self, event)
end

function resolve.medium(self, event)
end

function resolve.small(self, event)
end

local shape = {
```

2. Each function will start by unregistering its target object as a `clock` listener:

```
function resolve.large(self, event)
  self.parent:removeEventListener('clock', self)
end

function resolve.medium(self, event)
  self.parent:removeEventListener('clock', self)
end

function resolve.small(self, event)
  self.parent:removeEventListener('clock', self)
end
```

3. They will all finish by removing the asteroid that they're cleaning:

```
function resolve.large(self, event)
  self.parent:removeEventListener('clock', self)
  self:removeSelf()
end

function resolve.medium(self, event)
  self.parent:removeEventListener('clock', self)
  self:removeSelf()
end

function resolve.small(self, event)
  self.parent:removeEventListener('clock', self)
  self:removeSelf()
end
```

4. The two larger ones will spawn smaller ones, using the functions stored in the existing table:

```
function resolve.large(self, event)
  self.parent:removeEventListener('clock', self)
  for i=1,2 do
    local new = asteroids.medium(self.parent)
    new.x, new.y = self.x, self.y
  end
  self:removeSelf()
end

function resolve.medium(self, event)
  self.parent:removeEventListener('clock', self)
  for i=1,2 do
    local new = asteroids.small(self.parent)
    new.x, new.y = self.x, self.y
  end
  self:removeSelf()
end
```

5. Once that's done, each of the creation functions will attach the appropriate cleaning function to its new object as a `clock` handler. They will not register the object to receive `clock` events yet.

```
function asteroids.large(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  self.clock = resolve.large
  return asteroid(self, shape.large)
end

function asteroids.medium(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  self.xScale, self.yScale = 2/3, -2/3
  self.clock = resolve.medium
  return asteroid(self, shape.medium)
end

function asteroids.small(parent)
  local self = display.newImage(parent, "sprite/meteorSmall.png")
  self.clock = resolve.small
  return asteroid(self, shape.small)
end
```

6. These functions will also specify the point value of each asteroid created:

```
function asteroids.large(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  self.clock = resolve.large
  self.Points = 20
  return asteroid(self, shape.large)
end

function asteroids.medium(parent)
  local self = display.newImage(parent, "sprite/meteorBig.png")
  self.xScale, self.yScale = 2/3, -2/3
  self.clock = resolve.medium
  self.Points = 50
  return asteroid(self, shape.medium)
end

function asteroids.small(parent)
  local self = display.newImage(parent, "sprite/meteorSmall.png")
  self.clock = resolve.small
  self.Points = 90
  return asteroid(self, shape.small)
end
```

7. In the asteroid shared function, we'll attach another function to the new object as a collision listener:

```
local function clean(event)
end

local fadeIn = {time = 750, alpha = 0}
local rockFilter = {groupIndex = -2}
local function asteroid(object, shape)
  local body = {shape = shape, filter = rockFilter}
  physics.addBody(object, "dynamic", body)
  object:addEventListener('collision', clean)
  local push, theta = 2/3, math.random() * 2 * math.pi
```

8. In that collision function, we'll unregister the same collision listener (to prevent duplicate collision responses) and register the object to clean itself up on the next `clock` cycle. It doesn't need to add a `clock` handler, because that was attached when the object was created.

```
local function clean(event)
  local self = event.target
  self:removeEventListener('collision', clean)
  self.parent:addEventListener('clock', self)
end
```

9. Since game objects in this project can expect their parent to be the world object, each rock will pass an event to its parent indicating that it has been destroyed. The game object can listen to the world for these events and respond appropriately:

```
  self.parent:addEventListener('clock', self)
  self.parent:dispatchEvent{name = 'Destroy'; kind = 'rock', self}
end
```

10. It will also play an explosion animation at the point where the asteroid is when it gets hit:

```
  self.parent:dispatchEvent{name = 'Destroy'; kind = 'rock', self}
  explosion(self.parent, self.x, self.y)
end
```

11. This means that it needs access to the explosion module:

```
local physics = require "physics"

local explosion = require "explosion"

local asteroids = {}
```

Handling collisions with the player

Bouncing and shoving isn't how we want the rocks to interact with the player. We'll start dealing with this by adding a collision handler to the player object that will redispatch it as a semantic event, that is, one that has game meaning.

1. Open `player.lua` and add a new event listener after the world event code:

```
    self:setLinearVelocity(0, 0)
  end
  self:addEventListener('collision', self)
  function self:collision(event)
  end
  return self
```

2. This listener will simply send a new event to the object:

```
  function self:collision(event)
    self:dispatchEvent{name = 'Destroy'; kind = 'ship', self}
  end
  return self
```

3. Then add a listener for that new event, to add a suitable visual:

```
function self:collision(event)
  self:dispatchEvent{name = 'Destroy'; kind = 'ship', self}
end
self:addEventListener('Destroy', self)
function self:Destroy(event)
  explosion(self.parent, self.x, self.y)
end
return self
```

Now that the `player` object is ready to handle this collision event, we'll set up the game object to respond to the same event by resetting the `player` object in the level. Save `player.lua` and go back to `game.lua`.

4. Set up the scene's `willEnterScene` listener to register the scene to track `Destroy` events on the `player` object once it is created:

```
  self.Player = require "player" (self, self.World, self.
World:contentToLocal(display.contentCenterX, display.
contentCenterY))
  self.Player:addEventListener('Destroy', self)
  for i=1, 2 do
    local new = asteroids.large(self.World)
```

5. Near the top of the file, add a function to the scene object to process those events. If the destroyed object is described as a player, the game will queue up another function that will move the player back to the center and stop its motion.

```
  self:dispatchEvent{name = 'Jump'; x = x, y = y}
end

function scene:Destroy(event)
  if event.kind == 'ship' then
    local player = event[1]
    self:addEventListener('clock', reset)
  end
end

function scene:touch(event)
```

Collision handlers can't respond to a collision by moving an object, for the same reasons they can't destroy the object directly.

6. Because this function won't be attached directly to the player (in case the player code itself should need the `enterFrame` entry for something), the function will be created each time it needs to be applied, so that it can use the appropriate `player` object as an `upvalue`.

```
if event.kind == 'ship' then
  local player = event[1]
  local function reset(event)
  end
  self:addEventListener('clock', reset)
end
```

7. The function will first remove itself as a listener:

```
local function reset(event)
  self:removeEventListener('clock', reset)
end
```

The fact that the function needs to refer to itself is why it is created here as a named function and not as an anonymous one.

8. It will move its bound object to the center of the screen and set its velocity to `0`.

```
local function reset(event)
  self:removeEventListener('clock', reset)
  player.x, player.y = player.parent:contentToLocal(display.
contentCenterX, display.contentCenterY)
  player:setLinearVelocity(0, 0)
end
```

What did we do?

We used the collisions of these objects as a way to notify the objects of updates that they could not perform during the event response itself. We cleaned up one-shot listeners to prevent them causing errors later. We created new, different enemies in response to other ones being destroyed, and laid the groundwork for dealing with the player's death and losing the game, which we'll finish in the next section.

Recognizing kills and recording scores

The world-level interactions between the player, asteroids, and lasers are effectively complete at this point. The remaining tasks are for the game to track those elements that are not part of the world representation; accomplishments and challenges.

Getting on with it

To manage scores, first we'll need to set it when a game begins.

1. Open `game.lua` and find the `enterScene` handler, set the starting value for player score, and dispatch a game event:

```
Runtime:addEventListener('touch', self)
self.Score = 0
self:dispatchEvent{name = 'Score'; value = self.Score}
self:Start()
```

2. To trigger changes in the score, the game needs to track `Destroy` events for rocks, which it can receive from the world:

```
self:dispatchEvent{name = 'Score'; value = self.Score}
self.World:addEventListener('Destroy', self)
self:Start()
```

3. Before moving on, keep registration balanced by removing the listener in the `exitScene` handler:

```
function scene:exitScene( event )
  Runtime:removeEventListener('Tilt', self)
  Runtime:removeEventListener('Shake', self)
  Runtime:removeEventListener('touch', self)
  self.World:removeEventListener('Destroy', self)
  self:Stop()
end
```

4. Next, add another condition to the check in the game scene's `Destroy` handler, since this function will handle events from both sources:

```
    self:addEventListener('clock', reset)
  elseif event.kind == 'rock' then
    local rock = event[1]
  end
end
```

5. When a rock is destroyed, adjust the score and broadcast an event with the new score:

```
  elseif event.kind == 'rock' then
    local rock = event[1]
    self.Score = self.Score + rock.Points
    self:dispatchEvent{name = 'Score'; delta = rock.Points, value
= self.Score}
  end
```

Tracking lives

To make the game interesting, the player should have a finite number of lives that determine when the game is over.

1. Start by going back to the `enterScene` handler and setting the initial lives counts along with the starting score:

```
self.World:addEventListener('Destroy', self)
self.Lives = 2
self:dispatchEvent{name = 'Lives'; value = self.Lives}
self:Start()
```

2. Return to the `Destroy` event handler, and modify the branch for the player to adjust the number of lives and broadcast the change in an event:

```
    self:addEventListener('clock', reset)
    self.Lives = self.Lives - 1
    self:dispatchEvent{name = 'Lives'; delta = -1, value = self.
Lives}
  elseif event.kind == 'rock' then
```

3. If no lives remain available when the player is destroyed, the game is over; announce this to the game object with a `Game` state event:

```
  if event.kind == 'ship' then
    if self.Lives <= 0 then
      timer.performWithDelay(1,
        function(...)
          self:dispatchEvent{name = 'Game'; action = 'stop', Score
= self.Score}
        end
      )
    else
      local player = event[1]
      local function reset(event)
        self:removeEventListener('clock', reset)
        player.x, player.y = player.parent:contentToLocal(display.
contentCenterX, display.contentCenterY)
        player:setLinearVelocity(0, 0)
      end
      self:addEventListener('clock', reset)
      self.Lives = self.Lives - 1
      self:dispatchEvent{name = 'Lives'; delta = -1, value = self.
Lives}
    end
  elseif event.kind == 'rock' then
```

4. In response to that event, the game should pause the simulation and return to the menu screen:

```
scene:addEventListener('Game', scene)
function scene:Game(event)
  physics.pause()
  storyboard.gotoScene('splash')
end

local threshold = 1 / 6

function scene:Tilt(event)
```

Now that this event is being fired, there's one last piece of clean-up we can do properly. Save `game.lua` and open `player.lua`.

5. Before the constructor function returns the new object, register it to receive `Game` events from the game and respond to them:

```
  function self:Destroy(event)
    explosion(self.parent, self.x, self.y)
  end
  input:addEventListener('Game', self)
  function self:Game(event)
  end
  return self
```

6. In that handler, clean up the events that the player has registered for on the `game` object, since the `player` object is about to be disposed of.

```
  function self:Game(event)
    if event.action == 'stop' then
      world:removeEventListener('clock', self)
      input:removeEventListener('Thrust', self)
      input:removeEventListener('Yaw', self)
      input:removeEventListener('Fire', self)
      input:removeEventListener('Jump', self)
      input:removeEventListener('Game', self)
    end
  end
```

Displaying lives and the score

The last element to managing game progress is to make it visible to the player; no game will hold a player's interest if they can't see themselves getting closer to their goal. For this, we'll use a new group to hold the display elements, which we'll create along with the world.

1. All the appropriate events are already in place, so load and call the new module in `createScene` of `game.lua`, and install the resulting group in the scene:

```
  self.World = require "world" (self)
  group:insert(self.World)
  self.UI = require "interface" (self)
  group:insert(self.UI)
end
```

2. Next, create the `interface.lua` module file. Open it and frame in the function to create the new layer:

```
return function(game)
  local self = display.newGroup()
  return self
end
```

3. Add a text object to the layer to display the score:

```
  local self = display.newGroup()
  local score = display.newText(self, "", 10, 7, native.
systemFont, 14)
  score:setReferencePoint(display.TopRightReferencePoint)
  score.x, score.y = display.contentWidth - 10, 7
  return self
```

4. Register this text as a listener for `Score` events on the game, and give it a handler to process them:

```
  score.x, score.y = display.contentWidth - 10, 7
  game:addEventListener('Score', score)
  function score:Score(event)
  end
  return self
```

5. This handler can just display the new value in the text object, realigning the text as needed:

```
  function score:Score(event)
    local x, y = self.x, self.y
    self.text = tostring(math.floor(event.value))
    self:setReferencePoint(display.TopRightReferencePoint)
    self.x, self.y = x, y
  end
```

6. To keep the lives display contained, create a new group and make it a child of this one. Position it a little in from the top-left corner. This group will hold one child image visible for each life the player has:

```
    self.x, self.y = x, y
end
local lives = display.newGroup()
self:insert(lives)
lives.x, lives.y = 10, 7
return self
```

7. Register this group to receive `Lives` events from the game and handle them:

```
lives.x, lives.y = 10, 7
game:addEventListener('Lives', lives)
function lives:Lives(event)
end
return self
```

8. When an event is received, the group first shows as many children as the player has lives:

```
function lives:Lives(event)
  for i = 1, event.value do
    if self[i] then
      self[i].isVisible = true
    end
  end
end
```

9. If not enough images are available, it creates them:

```
  for i = 1, event.value do
    if self[i] then
      self[i].isVisible = true
    else
      local new = display.newImage(self, "image/life.png")
      new:setReferencePoint(display.TopRightReferencePoint)
      new.x, new.y = new.width * 1.1 * I, 0
    end
  end
```

10. Finally, it hides any life images already created that aren't required right now:

```
      self.x = self.width * 1.1 * i
    end
  end
  for i = event.value + 1, self.numChildren do
    self[i].isVisible = false
  end
end
```

What did we do?

We leveraged existing events to manipulate game-specific progress values, the player's accumulated score, and remaining lives. We created events to track those values, and a visible UI layer to display the values of those events.

Game over – wrapping it up

This project has covered a lot of ground. We practiced converting accelerometer input into a more readily usable form and interpreting it based on the player's orientation in the game world. We explored using several different aspects of the physics engine to control motion, including force, impulse, and velocity. We covered the basics of using collision filters to select which objects do and don't affect each other. We also extended our technique for calculating and tracking scores to cover game lives as well, which are core features in many game styles that future projects will use without extensive discussion.

And, of course, there were explosions!

Can you take the HEAT? The Hotshot Challenge

Add UFOs that come out semi-randomly; there's a suitable image in the original art files folder in the project pack. The UFOs should shoot red lasers in random directions at semi-regular intervals. Destroying a UFO grants the player an extra life. You should be able to do this with only one new code file and some modifications to the game scene code, although using two new files may yield a cleaner solution.

Project 5

Atmosfall – Managing Game Progress with Coroutines

If you're a long-time or low-level programmer, you may be accustomed to controlling the core of your game loop; if you're more used to recent game engines, you may have learned to juggle complex information about what actors in your game are doing at any given moment. In Corona, the combination of a separately-tracked physics engine and the versatility of Lua will, when used correctly, manage this information for you.

What do we build?

We'll explore this idea of managing progress through a game schedule by completing a scrolling shooter in the tradition of games such as *Xevious* or *River Run*, where the player maneuvers a ship around the screen, avoiding enemy fire and destroying enemy ships until he/she reaches a boss target. Enemies will include ships that fly various patterns across the screen, turrets that follow the scrolling background and turn to face the player's ship, and a boss at the end that takes many hits to destroy and moves in various directions, firing multiple weapons. The player's ship will be able to fire at points on the screen that the player touches.

What does it do?

The player maneuvers their ship around the screen as the ground scrolls under them from the top of the screen downward. As the scrolling background reaches certain points, enemies appear and fly or scroll around the screen, firing bullets as they go; the player must avoid the bullets as well as the ship itself while shooting back.

Like the *Deep Black* game, this project will be based on Corona's Box2D-based physics library. In this version, most objects will be "sensors", meaning they only detect collisions, and do not bounce off of each other or transfer momentum. We'll also use Box2D's collision filters so that we don't need to process enemy ships colliding with each other, or bullets hitting each other.

In the `TranslationBuddy` project, we got a taste of coroutines and how they can be used to bookmark a task that you're in the middle of and come back to it later. In this project, we'll take that concept much further, using coroutines to create scripted behavior that's carried out over time, which is the simplest part of what game players and developers usually refer to as AI. We'll create computer-controlled fighters that follow flight assignments which are predictable at run time, but easily customized by the designer.

Why is it great?

In addition to coroutines, we'll also use a Lua feature called **environments** to create a minimal language of very simple functions controlling intervals and enemy actions. Features like this can become useful in larger projects, where programmers and designers must collaborate on a project. In such projects, programmers are responsible for the code that carries out the actions which the enemies will take, but the decisions of what the enemies should do, when, and in what order, are made by game designers, and frequently have to be adjusted for best balance and fun. For this reason, it's good to let the designers edit these schedules and scripts themselves. Designers are usually not experienced programmers, although they often have a little knowledge of programming and scripting, so this simple language will make it much easier for the designers.

How are we going to do it?

For this project, we'll review the design, rather than creating it. While an indie programmer will often be designing their own games and then coding from their designs, any programmer at a studio is likely to be coding from a document given to them by a designer or design team. We will be managing game progress with the help of the following game coroutines:

- Founding the framework
- Moving the player
- Scheduling enemies
- Scripting behavior
- Controlling the boss
- Cleaning up and making the game playable

What do I need to get started?

First, open the file `design.txt` from the project pack and read through it, noting the [NYI] tags that indicate features still pending. In full development projects, this sort of status tracking will usually be carried out by a more complex database or dedicated tracking program, but even in small projects, a simple record of what has yet to be accomplished can be very useful.

At this point, the game has files describing various ships (a broad category which also includes the ground-based, immobile turrets) as well as the various weapons with which the ships are equipped, the bullets they fire, and the explosions when they land. It also includes code that handles things taking damage—including events that can be tracked by user interface elements or gameplay progress tracking, processing user input into commands, and a long background made up of large tiles. A splash screen is already completed and appears when the game is launched.

What the game still needs is actual level design. Enemies need to appear as time advances in the level, and carry out various plans of attack against the player. This means that there are two kinds of schedules required; the schedule of which enemies appear when, and the individual schedules of the enemies that dictate how each one flies and attacks after it is created. To make this happen, we'll not only create scheduling behaviors, but also modules that attach these behaviors and other suitable characteristics, such as orientation, to our predefined ships to make them into bosses and enemies.

This distinction between *units* in a game that have statistics and behaviors, and the *actors* that represent them in the game world, is very powerful for scaling projects up in complexity.

Once that's done, create a new project folder, `Atmosfall`, and copy the contents of the `version 0` folder in the project pack directory. You should be able to load this project into the simulator and advance the game past the splash screen, seeing a ship in the middle of a swamp background. In the simulator, you can click anywhere on the game screen and watch the ship fire double bullets at the point at which you click. You can also build the game for a device and watch the ship slide around the screen as you tilt the device. The prototype responds to user input and is ready to start adding the scheduling and AI enemy control.

Tracking progress through the level

Games like this typically trigger enemy appearances and events based on how far the background has scrolled past the screen. To focus on the challenges of the project, we'll import the background itself and just add the scrolling logic to issue events that track this progress.

Getting ready

You should have already copied the partly completed project from the `version 0` folder into your new project directory; if you haven't, do that now.

Getting on with it

We'll start by loading the new `marsh` background into the `Ground` layer of the game's view, instead of the blank rectangle that the project uses by default. Open the `game.lua` file and change the `createScene` function to load this module as the new background, as shown in the following code snippet:

```
local group = self.view
self.Ground = require "level.marsh"(group)
self.Mobs = display.newGroup()
```

Then we adjust the scale of the background to make it fit into the width of the screen:

```
self.Ground = require "level.marsh"(group)
local scale = display.contentWidth / self.Ground.width
self.Ground.xScale, self.Ground.yScale = scale, scale
self.Mobs = display.newGroup()
```

If you test the code at this point, you should see a static background filling the screen behind the player ship.

Sliding the background

Pinning a moving rectangle to another moving rectangle so that it doesn't slide too far and show the back side of the world requires a little bit of math, but we can offload this math onto Corona with a little ingenuity.

First, when the scene starts, we'll make sure that in the `willEnterScene` responder, the background is lined up, with its bottom edge along the bottom edge of the screen.

```
  physics.setGravity(0, 0)
  self.Ground:setReferencePoint(display.BottomCenterReferencePoint)
  self.Ground.xOrigin, self.Ground.y = display.contentCenterX,
display.contentHeight
  for _, coordinates in ipairs(walls) do
```

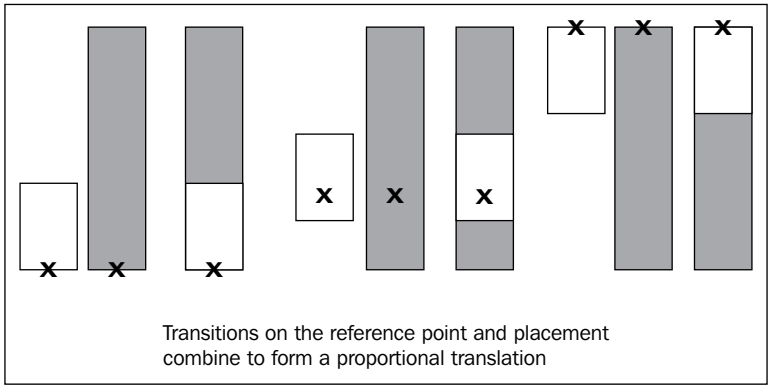
Before we can start the background moving, we need to know how long it's supposed to take. This could vary from level to level in a real game, so we will let the schedule we load provide this value (the file in question doesn't exist yet; creating it will be part of our next task, so make a note that it will need to return its total length) as shown in the following code:

```
physics.setGravity(0, 0)
self.Duration = require "level.marsh-enemies" (self)
self.Ground:setReferencePoint(display.BottomCenterReferencePoint)
```

Now that we know the desired length of the background scroll, we can start the transition in response to the `enterScene` event, once the scene has finished loading as shown in the following code snippet. This is where the magic happens and the explanation will follow once you've had a chance to scan it.

```
self.Exit = nil
local bounds = self.Ground.contentBounds
self.Ground.Pan = transition.to(self.Ground,
  {
    time = self.Duration * 1000;
    y = 0, yReference = bounds.yMin - bounds.yMax;
    onComplete = function(object)
      object.enterFrame, object.Pan = nil, nil
      end
  }
)
self.Lives = 2
```

The key here is that `transition.to` (and `transition.from`) can tween any value on any object that can be indexed, even though it usually gets applied to the visible properties of display objects. The `yReference` value has no visible effect on an object by itself; changing an object's `yReference` value changes what value its y position ends up being in its parent coordinates, but doesn't cause anything to move on the screen. However, when you combine this with continually adjusting the y position, it has the effect of changing the scale of the motion; setting the y position moves the object so that the point designated internally as its `yReference` value sits at the specified coordinate in its parent system.



Transitions on the reference point and placement combine to form a proportional translation

We also needed to know the distance to move the reference point in the group; the `marsh` module builds a group with its zero point, or origin, at its bottom. So eventually, we need to move the `yReference` value to a negative value equal to the functional height of the background. We figure this out from the height of its bounding rectangle.

If you want to test this out, you'll have to temporarily replace `self.Duration = require "marsh-enemies" (group)` with `self.Duration = 60`. To temporarily sub out values like this, I often create an end-of-line comment, making the line look like the following:

```
self.Duration = 60 -- require "level.marsh-enemies" (self)
```

Then, to replace the old code, I can just delete the part that says `60 --`. Once you've inserted this place-holder, you can test the code and you'll see the background crawl past. However, if you're using a tall profile device, such as an iPhone 5, you may see that the bottom edge of the background first creeps down across the empty bar at the bottom of the screen, before filling it completely, and at the end, it will expose a bar of black at the top as it creeps into place. This is because when Corona uses **letterbox** alignment, it only positions the reference frame for drawing to leave space at the edges; it doesn't actually crop out anything that was hanging over the edges of the screen. But we can do this fairly easily by adding a mask to the display stage, the way we did on scenes in *Project 3* to hide bits of overhanging text during transitions. To enforce this letterboxing globally throughout the program, we can add the mask in `main.lua`.

```
require "input"
display.currentStage:setMask(graphics.newMask("effect/masking-frame.
png"))
display.currentStage.maskX, display.currentStage.maskY = display.
contentCenterX, display.contentCenterY
local storyboard = require "storyboard"
```

Tracking the background progress

We need to track how far the schedule for the level will have advanced. We want this to be as fine-grained as we can manage, so we'll check the value as frequently as Corona will do, which is every frame. This means that an `enterFrame` listener is the logical vector which is shown in the following code snippet:

```
Runtime:addEventListener('enterFrame', self.Ground)
function self.Ground:enterFrame(event)
end
self.Lives = 2
```

For the schedule, we need to track time elapsed since the schedule starts. We'll make a note of the time when the scene begins, and post the `Progress` events that indicate what time the schedule has reached at, as shown in the following code:

```
self.Ground.Start = system.getTimer()
Runtime:addEventListener('enterFrame', self.Ground)
function self.Ground:enterFrame(event)
```

The other thing we want to track is the actual motion of the background. The first enemy we'll create is a turret, so if it doesn't appear to move in sync with the ground, the user will find that very distracting. This is a little trickier, since the ground's position and reference are both moving on the same sliding scale. Fortunately, there is another point that moves more predictably—the origin.

The **origin** of an object is whichever point is considered (0, 0) for placing that object's reference point. For most objects, it's fixed at the object's center; for groups, it's (0, 0) in the group's coordinates, wherever the group's children happen to be placed in relation to it. In our case, what we're really concerned with is that the object's origin is fixed with respect to its visible contents in a way that the reference point isn't. So if the group moves down by three pixels on the screen, you can say confidently that its `yOrigin` value also increased by three (assuming that it's not parented to a group with a different `yScale` value).

We can track the position of the ground's `yOrigin` value from frame to frame in order to determine how much it has visibly moved. We can include that information in the events we dispatch to supply the progress information as shown in the following code:

```
self.Ground.Start = system.getTimer()
self.Ground.oldY = self.Ground.yOrigin
Runtime:addEventListener('enterFrame', self.Ground)
function self.Ground:enterFrame(event)
  scene:dispatchEvent{name = 'Progress'; time = (event.time - self.
Start) / 1000; delta = self.yOrigin - self.oldY}
  self.oldY = self.yOrigin
end
```

Our `schedule` module will feed off these events to update its progress and trigger actions. First, however, we need something for the `schedule` module to do; schedules for levels will consist of spawning enemies at different intervals.

Constructing the enemy behavior

For the schedule to spawn new enemies, we need one to exist. We'll start with a simple one, that just moves in sync with the ground to start with. Once that works, we'll add tracking and weapon fire.

Getting on with it

Add a new file, `turret.lua`, and open it. This file will add `turret` behavior to a `ship` sprite and physics description specified in the starting project.

Creating an enemy

Add the basic description of the turret object's appearance and physics as shown in the following code:

```
local ship = require "ship.ship"
local category = require "category"
local groundFilter = {
  groupIndex = category.enemy;
}
return function(game, x, y)
  local self = ship.turret(game, game.Mobs.Ground, groundFilter)
  self.x, self.y = x, y
  self.bodyType = 'static'
  return self
end
```

This creates a turret that just sits perfectly still and does nothing, which isn't very interesting. The next thing we'll do is make it move in sync with the ground by following the `Progress` events that we added to the game in the last section as shown in the following code:

```
self.bodyType = 'static'
game:addEventListener('Progress', self)
function self:Progress(event)
  if self.y then
    self.y = self.y + event.delta
  else
    game:removeEventListener('Progress', self)
  end
end
return self
```

Since each `Progress` event contains the amount the screen moved, we can move the turret by the same amount. Now that this object has some basic behavior, it's time to start linking it into the game and attaching the scheduling mechanism. Save and switch to `game.lua`, and add a simple table before the event definitions using the following code:

```
local scene = storyboard.newScene()
scene.Spawn = {
  turret = require "turret";
}
function scene:createScene( event )
```

This means that we can call `game.Spawn.turret(game, x, y)` to create a new turret at `x, y` in the world for `game`. However, we're not going to call it directly. We'll create a schedule that contains functions to spawn enemies in a common, shared context (the game) to save the code having to contain the same values repeated an awful lot.

Creating a schedule

In order to have our turret appear at the right point, we'll create a schedule that spawns new enemies as the level progresses.

Getting on with it

Save `game.lua` for the moment and create a new file in the `level` folder called `marsh-enemies.lua`. This file will define a `schedule` module that's 60 seconds long, so that's the first thing we'll define using the following code:

```
return function(game)
  local duration = 60
  return duration
end
```

At this point, if you previously put a placeholder duration in `game.lua` to test the background scroll, you can revert that to the code that uses this file.

Next we'll use the `schedule` function (from a module that isn't created yet) to start our custom `schedule` function against the current game. This will take a few steps to make it fully clear, but showing you how it will be used in the following code should help you see why this setup is worth engineering:

```
local duration = 60
schedule(game,
  function()
```

```
      at (0.3) spawn.turret(50, -20)
    end
  )
  return duration
```

Now here's the catch—the `at` and `spawn.turret` functions haven't been defined anywhere yet, and they won't be made local in this file or defined as globals, even though this function uses them as global. Our `schedule` function will create them in a custom environment.

So for each `schedule` function, we'll create an environment that contains simplified actions on the game the schedule is for, and use it for the function that defines the schedule. We'll combine this with making the schedule function part of a coroutine, so that it can suspend itself, such as when it is waiting for a particular time to come up in the schedule.

So, before moving on, load the module you're about to create into `marsh-enemies.lua` using the following code:

```
local schedule = require "schedule"
return function(game)
  local duration = 60
```

Building a schedule framework

Save `marsh-enemies.lua` and create the file `schedule.lua` at the top of your project. This file won't actually be very long. The core is a function that starts each newly created coroutine, attaching the environment supplied to the schedule, running that schedule until it's complete, and finally disconnecting the schedule from the game so that it won't throw errors or take up processing time as shown in the following code:

```
local function bind(game, listener, actions, schedule)
  setfenv(schedule, actions)
  schedule()
  game:removeEventListener('Progress', listener)
end
```

The rest of the module will be a function that does the work of setting it up. It'll create an environment that contains bridge actions to the main game, start a coroutine using our glue function, and start that coroutine with the `schedule` function and environment. This coroutine will wake up every time the `game` object receives a `Progress` event to see if there are any enemies it needs to spawn, create those required, and go back to waiting.

We'll start by creating a blank environment and our coroutine:

```
  game:removeEventListener('Progress', listener)
end
return function(game, schedule)
  local actions = {}
  local self = coroutine.wrap(bind)
end
```

Unlike `coroutine.create`, `coroutine.wrap` returns a function that resumes the new coroutine each time it's called. It's usually a little more convenient, but be a little careful with `coroutine.wrap` because if any error is thrown inside the coroutine, it will bubble right up and affect the code calling the `resume` function.

We'll then connect the new coroutine to be resumed for each `Progress` event sent to the game using the following code:

```
  local self = coroutine.wrap(bind)
  game:addEventListener('Progress', self)
end
```

We'll create a listener that stops feeding new `Progress` events into the `schedule` module when the game ends, such as when the player loses all of his or her lives as shown in the following code. This effectively terminates the schedule; there's no way anymore to resume it and it will get garage-collected.

```
  game:addEventListener('Progress', self)
  local function close(event)
    if event.action == 'ended' then
      game:removeEventListener('Progress', self)
      game:removeEventListener('Game', close)
    end
  end
  game:addEventListener('Game', close)
end
```

Then, we'll start the coroutine with its schedule and environment, as well as the information it needs to clean itself up, and return the new coroutine in case the calling code has some use for it:

```
  game:addEventListener('Game', close)
  self(game, self, actions, schedule)
  return self
end
```

Building the scheduled actions

Of the two functions we've described, the `at` action is the simpler one. It checks the elapsed time in the `schedule` module; if it's not enough, it yields to keep waiting, but if its designated time has arrived, it returns from its loop and lets the schedule advance. This means the code is very straightforward as follows:

```
local actions = {}
function actions.at(time)
  repeat
    local progress = coroutine.yield()
  until progress.time >= time
end
local self = coroutine.wrap(bind)
```

Calling the function `at` may seem a little strange, but it allows the `schedule` calls to read much more like normal language. We could have called the function `waitUntil` and written the function waiting for it on another line, but Lua's loose syntax allows us to use this very compact format.

The `spawn` functions are a little more complex. We could simply build them all like the following:

```
function actions.spawn.turret(x, y)
  return game.Spawn.turret(game, x, y)
end
```

However, since each one would follow the same pattern, we can use another pattern based on metatables and the `__index` lookup, the **self-populating table**. `__index` on a table's metatable is only used when the requested key isn't in the original table; this means that the function that creates the requested value can store it in the table, and next time, it will simply be retrieved from the table instead of being looked up in the `__index` table again. This makes the spawn action family easy to summarize in one block.

```
actions.spawn = setmetatable({},
  {
    __index = function(t, name)
      local function cue(...)
        return game.Spawn[name](game, ...)
      end
      t[name] = cue
      return cue
    end
  }
)
local self = coroutine.wrap(bind)
```

That's actually all there is to the schedule system! You can test it out and watch the turret appear in the upper-left and slide-down corner across the screen, mindlessly facing right and doing nothing.

Bringing an enemy to life

Making the turret work requires only two main steps. The first is to have it track the player. Since the `game` object keeps a reference to the `Player` object, this boils down to basic trigonometry.

Open `turret.lua` and add a line to the `Progress` handler as shown in the following code:

```
    if self.y then
      self.y = self.y + event.delta
      self.rotation = math.deg(math.atan2(game.Player.y - self.y,
game.Player.x - self.x))
    else
```

The arctangent of the `y` and `x` distances gives the facing direction from the turret to the player, which we can use as the rotation angle to make the turret point at the player. If you try the code again you should see the turret stay pointed at the player as it scrolls down and the player moves. The last step is to give the turret a weapon. While we have the `turret` file open, have it start firing as soon as it is created:

```
  end
  self.Weapons.AntiAir:start(self, 100, 0)
  return self
```

This depends on the turret having a weapon called `AntiAir`, which in this case was already created in the existing partial code.

The turret is complete for the time being. If you test the code, it should shoot at the player, and bullets fired at it should vanish when they hit it. Currently, however, nothing gets destroyed no matter how many bullets hit it.

What did we do?

A lot of stuff happened in this section! Using coroutines to track a continuously advancing schedule of new enemies, using Lua environments to wrap up some complicated actions in some simple wrappers, and adding an advancing series of progression data to drive everything else.

What else do I need to know?

Environments are a powerful feature of Lua; each function has an associated environment which is just a table linked to that function. Whenever a function needs to use a global, it looks up the global name as a string key in that table. By changing a function's environment, we can give it access to a totally different set of global functions. The real power here will come from the fact that the environment that we link to our schedule will be stocked with functions that use the normal environment, and can therefore take actions whose particulars are hidden from the code using them.

Lua creates a default environment to link to its code when it's started, containing all the standard global functions, and stores a link to that environment in that environment under the name `_G`, which you've probably used already.

Scripting behavior

An enemy that just sits and shoots predictably quickly stops presenting much of a challenge. Our next enemy will be capable of steering around the screen according to a predesigned path, but since the path will be managed through a schedule similar to the one we use to spawn enemies, a different path can be given to each enemy.

Getting ready

This new enemy will be a fighter craft equipped with a single forward-facing machine gun that can be turned on and off. It can face a given direction, set its speed, and adjust its facing over time to create more naturally curved paths. We'll use the third entry in `ship/sprite.lua` to represent this ship.

Getting on with it

Prepare a new file in the top level of the project called `dogfighter.lua`. It will start much like the `player.lua` and `turret.lua` files. Notice that this function takes an extra argument—the function that describes its orders as shown in the following code:

```
local ship = require "ship.ship"
local category = require "category"
local enemyFilter = {
  groupIndex = category.enemy;
}
return function (game, x, y, path)
  local self = ship.fighter(game, game.Mobs.Air, enemyFilter)
```

```
  self.x, self.y = x, y
  return self
end
```

Because the fighters will face down by default, we'll rotate it in the opposite direction from the player. To keep the shadows and highlights on the sprite still appearing to come more or less from the same side, we'll reverse the `y` scale.

```
  self.x, self.y = x, y
  self.rotation = 90;
  self.yScale = -1;
  return self
end
```

We'll set a `Speed` value that will hold the ship's desired scalar velocity, without regard to which direction it's going in or how that velocity will be broken down into the `x` and `y` speeds:

```
  self.rotation = 90;
  self.yScale = -1;
  self.Speed = 0
  return self
end
```

We'll use a `plan` module, to be created next, to generate a coroutine that will carry out the requested orders, similar to the `schedule` module:

```
  self.Speed = 0
  local plan = require "plan" (self, path)
  return self
```

Finally, we'll start a timer to update this coroutine according to the passage of time, and store this timer so that we can stop it when the ship is destroyed or removed:

```
  local plan = require "plan" (self, path)
  self.Guidance = timer.performWithDelay(1000/application.fps, plan,
0)
  return self
```

Writing a ship control script

To get a notion of what commands our `plan` module needs to support, let's lay out a control script first. The ship will start by facing down across the screen, move forward at a set speed, and start firing as it travels down. After a second, it will veer off to the left and stop firing. This function will be laid out in the `marsh-enemies.lua` module, so save `dogfighter.lua` and reopen that module to add the following code:

```
    function()
      at (0.3) spawn.turret(50, -20)
      at (0.5) spawn.defender(140, -10,
```

```
        function ()
          face(90)
          go(100)
          fire("MachineGun", 50, 0)
          after (1) turn(60, 0.75)
          release("MachineGun")
        end
      )
    end
```

So, this gives us at least six functions to support:

- `face()` will instantly set the ship to face the specified direction.
- `go()` will set the ship's forward velocity to the specified speed.
- `turn()` will adjust the ship's facing by the specified angle relative to its current facing, over the specified amount of time.
- `after()` will wait until the specified amount of time has passed since `after()` was called, then proceed. It's not quite the same as `at()` in the level schedule, because that waits for specific intervals from the start of the level.
- `fire()` and `release()` will start or stop the specified weapon. The same key should be used that was used to identify the template in the ship's `Weapons` table.

The `dogfighter` module uses the `plan` module to support these functions, so save `marsh-enemies.lua` and create the new module, `plan.lua`, in the top level of the project. This file will start off with a skeleton similar to that of `schedule.lua`, as shown in the following snippet, with which it has a great deal in common:

```
local function bind(path, actions)
end
return function(self, path)
end
```

The plan will have the same sort of glue function, which will attach a set of actions to the `plan` function in the same way:

```
local function bind(path, actions)
  setfenv(path, actions)
  path()
end
```

However, where schedules are intended to run for a fixed amount of time, a ship's orders need to be carried out as long as it's alive, which could be highly variable. But since we expect plans to run off of a timer that will repeat indefinitely, this makes them more uniform to cancel once the `plan` function is exhausted. We just wait for one more firing of the timer so that we can get the timer source to cancel it.

```
  path()
  local finalize = coroutine.yield()
  timer.cancel(finalize.source)
end
```

The code to start a new plan from the glue function and arguments will be almost identical to that of `schedule.lua`:

```
return function(self, path)
  local actions = {facing = self.rotation}
  local plan = coroutine.wrap(bind)
  plan(path, actions)
  return plan
end
```

The ship control code will be able to refer to `facing` as a global variable to determine the angle at which the ship is pointing, although setting this global will not turn the ship. This is convenient for scripts that might want smarter behavior.

Defining ship actions

The easiest actions to implement will actually be the fire control ones:

```
local actions = {facing = self.rotation}
function actions.fire(name, xAim, yAim)
end
function actions.release(name)
end
local plan = coroutine.wrap(bind)
```

We'll put a sanity check in just so that if a designer (or programmer) misspells a weapon name or otherwise tries to fire a non-existent weapon, it doesn't crash the ship's entire plan:

```
function actions.fire(name, xAim, yAim)
  if self.Weapons[name] then
    self.Weapons[name]:start(self, xAim, yAim)
  end
end
```

We'll take a similar approach with the `release` function, but as a convenience, it can be called without any name to stop *all* firing weapons:

```
function actions.release(name)
  if name and self.Weapons[name] then
    self.Weapons[name]:stop()
  else
    for name, weapon in pairs(self.Weapons) do
      weapon:stop()
    end
  end
end
```

The `go` function is one of the functions that I was referring to when I mentioned that some functions get simpler when the base ship images are pointed to the right. It also stores the scalar speed to facilitate turning functions:

```
local actions = {facing = self.rotation}
function actions.go(speed)
  local angle = math.rad(actions.facing)
  self:setLinearVelocity(speed * math.cos(angle), speed * math.
sin(angle))
  self.Speed = speed
end
function actions.fire(name, xAim, yAim)
```

The Corona `rotation` properties are measured in degrees, but Lua `math` functions return and expect angles in radians. The `math.deg` and `math.rad` functions help plug this gap.

Finally, we get to use `turn()`. This is a slightly more complicated function because it keeps control of the ship until the turn is complete. For convenience, it expects durations in seconds, not milliseconds, as shown in the following code:

```
  self.Speed = speed
end
function actions.turn(angle, duration)
  duration = duration * 1000
end
function actions.fire(name, xAim, yAim)
```

It needs to know how much time has passed since the last time it was checked, so it starts by noting the time it started:

```
function actions.turn(angle, duration)
  duration = duration * 1000
  local start = system.getTimer()
end
```

The function will continue reducing the duration by the elapsed time whenever it gets notified of time passing, until the complete duration has passed:

```
    local start = system.getTimer()
    while duration > 0 do
      local elapsed = coroutine.yield()
      elapsed, start = elapsed.time - start, elapsed.time
      duration = duration - elapsed
    end
  end
```

To turn the ship smoothly, it will see what portion of the turn duration has just elapsed (maxing out at *all of it*), and turn by that large a slice of the remaining angle:

```
    while duration > 0 do
      local elapsed = coroutine.yield()
      elapsed, start = elapsed.time - start, elapsed.time
      local wedge = angle * math.min(1, (elapsed / duration))
      actions.face(actions.facing + wedge)
      duration = duration - elapsed
    end
```

Finally, it will reduce the remaining angle to turn by the amount it just turned by:

```
      actions.face(actions.facing + wedge)
      angle = angle - wedge
      duration = duration - elapsed
    end
```

Adding a new ship to the level

Open `game.lua` and add the `dogfighter` module to the list of elements the game can create:

```
scene.Spawn = {
  turret = require "turret";
  dogfighter = require "dogfighter";
}
```

If you test the code at this point, you will see a ship fly down across the screen and bank, although it should not fire yet.

Adding weapon fire

Save and move to the file `ship.lua` in the `ship` folder of your project directory. Find the section where it declares the `fighter` description, and fill a new entry into the `Weapons` table:

```
    MaxHealth = 1;
    Weapons = {
      MachineGun = weapon.MachineGun,
    };
  },
  dreadnought = ship {
```

If you test the code again, the enemy ship should fire as it comes down from the top of the screen.

What did we do?

We expanded the principles used for the `schedule` module to run actions in an even more specific context by using a single ship as the basis for the environment applied to the new plan. This system opens up a lot of options for producing varied ship behaviors to challenge and engage the player. Ships can fly in formation, appear in waves, and use various fire patterns to spray the screen with bullets.

What else do I need to know?

There's one important caveat to using functions with environments like this, in Lua, functions are actually objects; this means that if you set different environments on a function, the environment more recently set is used. And if you have multiple references to a function, setting the environment on one of them changes all of them. This means that if you try to use the same function as a plan for more than one ship at once, very strange things may happen, as if all the guidance systems were interfering.

The easiest way to deal with this is to make plan factories, functions that return a new copy of a function each time they're called. Each such closure, or instance of function code, has its own upvalues and its own distinct environment. By the end of the project, we'll have constructed several of these.

Controlling the boss

No game like this is complete without an awe-inspiring boss craft or base that offers a more complex fight. Because boss crafts don't typically fly around in circles, a new action is called for in `plan.lua`; `hover`. It's very similar to `turn` in its implementation, so we won't cover it in too much detail.

Getting on with it

First, open `plan.lua` and add the hover function to the environment:

```
  self.Speed = speed
end
function actions.hover(xMove, yMove, duration)
  duration = duration * 1000
  local start = system.getTimer()
  while duration > 0 do
    local elapsed = coroutine.yield()
    elapsed, start = elapsed.time - start, elapsed.time
    local fraction = math.min(1, (elapsed / duration))
    local xDrift, yDrift = xMove * fraction, yMove * fraction
    xMove, yMove = xMove - xDrift, yMove - yDrift
    self.x, self.y = self.x + xDrift, self.y + yDrift
    duration = duration - elapsed
  end
end
function actions.turn(angle, duration)
  duration = duration * 1000
```

Creating the boss object

Now that the AI script actions are complete, we'll create the file that will accept these actions and advance the game when a boss is destroyed. Make a copy of `dogfighter.lua` in the top level of the project and name it `boss.lua`. The following are the steps to create a boss object:

1. The `module` function will accept a flight `plan` function like the `dogfighter` module does, but also accepts a function that dictates what will happen when the boss is defeated:

   ```
   return function (game, x, y, path, advance)
     local self = ship.dreadnought(game, game.Mobs.Air, enemyFilter)
   ```

2. Record the `advance` function in the boss's table so that we can retrieve it and run it when the boss dies:

   ```
     self.Speed = 0
     self.Advance = advance
     local plan = require "plan" (self, path)
   ```

Using a supplied function gives us choices about how bosses appear. For instance, you could make a level where a boss appeared halfway through, and its defeat started a new schedule to proceed through the rest of the level.

Making the boss module available

In order for the schedule to be able to spawn a new boss object, the game's spawn table needs a reference to the module. Open `game.lua` and add a new line to the table near the top of the file.

```
scene.Spawn = {
  turret = require "turret";
  dogfighter = require "dogfighter";
  boss = require "boss";
}
```

Driving the boss's behavior

To launch the boss at the end of the level, we'll modify the level schedule. Open `marsh-enemies.lua` and find the end of the function being passed to the schedule.

1. We'll add an entry a second or two before the schedule runs out to create the `boss` object:

```
      at (58) spawn.boss(160, -40, orbit(), bossOneDestroyed)
    end
  )

  return duration
```

2. Because it's easy in this case, we'll specify what happens when the boss is defeated first:

```
end
local function bossOneDestroyed(self, game)
  game:dispatchEvent{name = 'Game'; action = 'ended', outcome =
'won'}
end
return function(game)
```

3. Then we'll create a function that returns a set of flight orders moving in a figure-eight pattern; first, the boss will move down from its spawning point at the top of the screen and slide over to one corner of its loop:

```
local function orbit()
  return function()
    hover(0, 100, 2)
    after (1.5) hover(55, 0, 1.5)
  end
end
local function bossOneDestroyed(self, game)
```

4. Then, its flight plan will go into a loop that will last until it is destroyed, moving down, then up and over, and then the same on the other side.

```
return function()
  hover(0, 100, 2)
  after (1.5) hover(55, 0, 1.5)
  while true do
    after(1) hover(40, 50, 1.5)
    after(1) hover(-150, -50, 4)
    after(1) hover(-40, 50, 1.5)
    after(1) hover(150, -50, 4)
  end
end
```

5. As the boss moves, it will switch some of its guns on and off as it changes direction. Because these calls are repetitive, the flight plan can define reusable functions for them.

```
after (1.5) hover(55, 0, 1.5)
local function crossFan()
  fire(1, 50, 80); fire(2, 50, -80)
  release(3); release(4)
  fire(5, 50, 13); fire(6, 50, -13)
end
local function focusForward()
  release(1); release(2)
  fire(3, 48, -15); fire(4, 48, 15)
  release(5); release(6)
end
while true do
```

This ability to keep using loops, functions, and other programming semantics inside a sandbox like this makes it especially powerful.

6. Once it calls those functions appropriately, the flight plan is complete.

```
while true do
  focusForward()
  after(1) hover(40, 50, 1.5)
  crossFan()
  after(1) hover(-150, -50, 4)
  focusForward()
  after(1) hover(-40, 50, 1.5)
  crossFan()
  after(1) hover(150, -50, 4)
end
```

Handling the boss's defeat

The default `onDestroy` handler falls short in two regards where the boss is concerned. It needs to end the level (as determined by the boss's advance function), and frankly, one little explosion isn't impressive enough for a boss's victory. The following are the steps to handle the boss's defeat:

1. Open `ship.lua` and frame in an `onDestroy` override for the `dreadnought` entry:

```
dreadnought = ship {
  sprite = chassis [4];
  MaxHealth = 1000;
  onDestroy = function(game, event)
    local self = event.unit
  end;
  Weapons = {
```

2. Most of the functions are still the same as the default `onDestroy` function, so we'll start by bringing them in. However, we'll hold off on removing the `boss` sprite as shown in the following code:

```
  onDestroy = function(game, event)
    local self = event.unit
    timer.cancel(self.Guidance)
    for id, weapon in pairs(self.Weapons) do
      weapon:stop()
    end
  end
```

3. First, we'll start a repeating timer to set off explosions continuously on top of the `boss` sprite:

```
    local w, h = self.width / 2, self.height / 2
    local chainReaction = timer.performWithDelay(100,
      function(event)
        explosion(self.parent, self.xOrigin + math.random(-w,
w), self.yOrigin + math.random(-h, h))
      end,
      0
    )
```

4. Then, we'll start a one-use timer that will clean things up. It will stop the endless explosions, remove the boss, and trigger the boss's `Advance` function.

```
    self.Destruction = timer.performWithDelay(1000,
      function(event)
        timer.cancel(chainReaction)
        self:Advance(game)
```

```
            self:removeSelf()
          end
        )
```

Test things out! Once you make it past the other enemies and dodge the boss's guns, you'll see the boss going down in a blaze of glory. Having your character be invulnerable during development makes things like this much easier to test out.

What did we do?

We expanded the selection of actions available to the enemy's AI code. We created a new object type with a continuously repeating set of controls and internal logic to its actions. We also used one timer to regulate another, in order to create a visually interesting effect with a fixed duration.

What else do I need to know?

If you want to create functions for your control coroutines, they need to be defined inside the function that contains the instructions. Otherwise, they'll be created without the special environment that provides their instructions, and they won't respond properly.

Cleaning up and making the game playable

There are three issues left before the game fulfills its design:

- Filtering collisions based on `groupIndex` alone doesn't prevent bullets from hitting each other, or allow the player ship to fly over turrets without colliding with them.
- One enemy before the boss is not going to provide a satisfying challenge. The level should provide several fighters and turrets to challenge the player over the level's course.
- The player can't actually lose lives or fail the game. This isn't a bug, but a feature that's deliberately been disabled until the end, in order to make the game easier to test.

Getting on with it

In order to control collisions more accurately, we'll use an additional feature of collision filters—**category masking**. This feature allows you to specify up to 16 categories that an object can belong to, as well as which of those categories the object will collide with. An object is not required to collide with objects from its own category. If two objects overlap and only one of them can collide with the other, no collision takes place and the two objects pass through each other.

The category and mask for an object are defined as bitmasks, numbers where each bit in the representation is treated as a separate on-off value. The easiest way to write these values in your code is as hexadecimal integers: `0x0001`, `0x0002`, `0x0004`, and so on. You can then add these values together to make composite bitmasks.

We want the player ship to collide with enemy bullets and enemy ships; it is already prevented from colliding with its own bullets or any friendly ships the game could contain by sharing the same negative `groupIndex` value. We will use the bit value `0x0001` to signify ships, and the bit `0x0002` to signify bullets; to keep this straight and clear, we'll define these in a file. Open the file `category.lua` and add three new entries to the table being returned:

```
return {
  ship = 0x0001,
  bullet = 0x0002,
  all = 0xFFFF;
  general = 0,
```

Save this file and open `player.lua`; set the filter used by the `player` object to create an object that belongs to the `ship` category and collides with objects from either the `ship` or `bullet` category.

```
local playerFilter = {
  groupIndex = category.player;
  categoryBits = category.ship, maskBits = category.ship + category.
bullet
}
```

The `dogfighter.lua` and `boss.lua` files get the same treatment; these flying ships can collide with the player's ship (destroying them both) or the player's bullets, as shown in the following code:

```
local enemyFilter = {
  groupIndex = category.enemy;
  categoryBits = category.ship, maskBits = category.ship + category.
bullet;
}
```

Make this change to both files using the following code, then open `turret.lua`. This one requires slightly different treatment; player bullets can destroy the turrets, but the player's ship can fly freely over them without either being destroyed.

```
local groundFilter = {
  groupIndex = category.enemy;
  categoryBits = category.ship, maskBits = category.bullet;
}
```

Notice that the turret's mask doesn't include other ships; this means that even if those ships are set to collide with ships (which the turrets still count as), they won't, since one-way collisions don't count.

Once you've saved this change, open `weapon.lua`. This one is slightly different because each weapon is owned by a particular group, and fires bullets that belong to that group, not hitting other objects from it. Find the anonymous function that is returned from the internal local function `weapon`, where the filter for each new weapon is defined from the group index used to create it as shown in the following code:

```
self.filter = {
  groupIndex = groupIndex;
  categoryBits = category.bullet, maskBits = category.ship
}
return self
```

Collision filters are now set up; bullets don't collide with each other, ships only collide if they're on different sides, and turrets can be shot but don't directly prevent flyovers.

Adding challenges to the level

Adding new enemies to the schedule can be simple or complex. Creating a turret is as simple as specifying a position, and optionally waiting for a specified time by using the `at (time)` function. As we've seen, creating a fighter is still simple, but requires supplying a function that lays out the ship's flight plan. Fighters and their patterns constitute the main attraction of a game like this, so we'd probably like quite a few, flying in variable patterns. Rather than copying and pasting the flight plan code for each spawn command, we can produce variations on common flight patterns by using function factories.

A function factory is another function that takes some arguments of the programmer's choice, and produces another function that doesn't have to run right away, but, when it does run, it produces a common behavior adjusted by the arguments that were used to create it. Each time the factory is used, it produces a new version of the desired behavior function, which also means that each of these functions can have its own environment (which is important for our ship controllers). We've already created one factory, actually, in the form of the `orbit` function that controls the boss. Now we'll produce a few more for a variety of flight behaviors.

In `level/marsh-enemies.lua`, create a new function, `cross`. This function will produce variable versions of the flight plan used by the `dogfighter` module we already created, that flies straight down across the screen, firing, then turns off the gun and veers off to one side before leaving the screen. The `cross` function will take one argument, the angle to turn left or right when it changes course as shown in the following code:

```
local schedule = require "schedule"
local function cross(angle)
  return function()
  end
end
local function orbit()
```

The body of the returned function will resemble the anonymous function supplied to our sample `dogfighter` function. You can enter it again or cut and paste the previous code; note that the fixed turn, 60, is replaced with the enclosing `angle` argument.

```
local function cross(angle)
  return function()
    face(90)
    go(100)
    fire("MachineGun", 50, 0)
    after (1) turn(angle, 0.75)
    release("MachineGun")
  end
end
```

Trim out the existing entry for the dogfighter at the `0.5` seconds, and add in a few lines that use this function about halfway through the level:

```
at (0.3) spawn.turret(50, -20)
at (26) spawn.dogfighter(140, -20, cross(-75))
at (27.5) spawn.dogfighter(180, -20, cross(75))
at (29) spawn.dogfighter(140, -20, cross(-75))
at (58) spawn.boss(160, -40, orbit(), bossOneDestroyed)
```

You can test this, if you like, to see how the fighters come out one by one.

We're going to add new patterns for a few more fighters, but first let's drop in a couple more turrets over the course of the map.

```
at (0.3) spawn.turret(50, -20)
at (12) spawn.turret(245, -20)
at (26) spawn.dogfighter(140, -20, cross(-75))
at (27.5) spawn.dogfighter(180, -20, cross(75))
at (29) spawn.dogfighter(140, -20, cross(-75))
```

```
        at (35) spawn.turret(275, -20)
        at (42.5) spawn.turret(40, -20)
        at (58) spawn.boss(160, -40, orbit(), bossOneDestroyed)
```

We'll add two more flight plan factories using the following code: one that just flies right across the screen at a chosen angle, and one that flies in a wide arc and travels back the way it came. Both start and stop shooting during their course.

```
      release("MachineGun")
    end
  end
  local function strafe(angle)
    return function()
      face(angle)
      go(100)
      fire("MachineGun", 50, 0)
    end
  end
  local function swoop()
    return function()
      face(90)
      go(100)
      after (0.1) turn(30, 0.75)
      fire("MachineGun", 50, 0)
      turn(-240, 3)
      release("MachineGun")
      turn(30, 0.75)
    end
  end
  local function orbit()
```

We can spice things up after the first turret with a couple of the simpler flyers using the following code:

```
        at (0.3) spawn.turret(50, -20)
        at (6.5) spawn.dogfighter(330, -15, strafe(120))
        at (9) spawn.dogfighter(-10, -15, strafe(60))
        at (12) spawn.turret(245, -20)
```

An advantage of presenting the flight plans as functions in Lua is that they can contain loops and choices like any code. We'll generalize the fliers using the `swoop` pattern using a `for` loop:

```
at (12) spawn.turret(245, -20)
for i=1,3 do
  at (20 + i) spawn.dogfighter(180, -20, swoop())
end
at (26) spawn.dogfighter(140, -20, cross(-75))
```

Save `marsh-enemies.lua` and try running through the whole level and see how the different enemies appear. Right now you're still invincible!

Enabling finite lives

In this particular case, we'll take the easy way out. The code to subtract lives when the player is hit was in the code when we started the project, and was turned off to make debugging and testing easier. This code is extremely similar to a module that was created in the *Deep Black* project, which is why it isn't covered in detail here.

Open `game.lua` and find the commented line that registers the game object to track `Destroy` events on the player and add the following code:

```
  local Player = player(self, display.contentCenterX, display.
contentCenterY)
--Player:addEventListener('Destroy', self)
  self.Player = Player
```

Uncomment the line to allow the game to remove a life or end the game whenever the player's ship is destroyed:

```
  local Player = player(self, display.contentCenterX, display.
contentCenterY)
  Player:addEventListener('Destroy', self)
  self.Player = Player
```

It's often advantageous to do things like this, creating systems early to make sure they work, then disabling them while they're inconvenient. It's time to try out the game! Careful, it's pretty hard.

Game over – wrapping it up

This was a serious endeavor! This system has several complex elements and uses all three major elements of advanced Lua: metatables, environments, and coroutines. It combines the special controls of smartphones with a classic retro style of arcade game play.

Can you take the HEAT? The Hotshot Challenge

A lot of these games have a special backup weapon of some kind. You can combine the `weapon` objects of this game with the multitouch techniques from *Project 8*, *The Beat Goes On* to create a bomb that starts charging when you touch your ship with one finger while already firing with another. Once both fingers have been held on the screen long enough, the bomb goes off around the point where the original finger is targeting the screen and deals damage to all enemies nearby. Try making an effect like a targeting circle that zooms in on the space being targeted as the bomb gets closer to falling.

Project 6

Predation – Creating Powerful Visuals from Simple Effects

Corona is a two-dimensional engine that currently focuses on mobile development, which means that the devices it targets are limited in their graphical processing power compared to desktop and laptop computers. Nonetheless, most of these devices have enough graphics power to make them comparable to, say, 32-bit game consoles, and they are easily capable of compositing images, layering elements, and clipping images or groups to a mask. While the engine doesn't currently support the dynamic creation of masks or sprite sheets, some creative application of the effects that it does support allows for a wide range of effects to be created with very little code.

What do we build?

For this short project, we will take a functionally complete game and apply some visual polish to it. This game, **Predation**, is a relative of the classic arcade game *Missile Command*, where zombies that explode into pools of diseased, corrosive blood when shot, are substituted for the explosive missiles of the original. However, currently the graphics are a bit crude; the zombies simply vanish when the blood pools eats them away, and the blood pools are perfectly round, flat pools of red color.

What does it do?

In this particular project, we have a game that is already feature-complete. You can play all the way through it, tapping the screen to dissolve marauding zombies in to fast-evaporating pools of caustic goo and turning the zombies into new pools of goo in the process. Instead of adding new functionality, we're going to focus on **plussing**, or **juicing**, the game before release. These are terms used by some game developers to refer to the process of looking at a game or other work in progress and looking for the parts of it that can most easily be made plus one or plus ten percent to improve value and appeal for the effort, or pumping juiciness into the game's presentation.

Zombie games tend to be aimed at a market that's interested in gore and explosions. While going over the top in this regard would be both expensive and potentially draw us some complaints, we can add some eye-catching appeal with only a little work. Ideally, the zombies should appear to dissolve or break up as the goo washes over them; we can simulate this by using a sort of tone effect to erase them a bit at a time. We'll also liven up the blood pools with some texture, liquid shine, and some variation in the outline.

Why is it great?

This project will showcase the extreme ease of coding most visual effects and transitions in Corona. While to some extent this is offset by the need to create mask images and effect templates, the effects images used for this project were created using the free image software **GIMP** (the **GNU Image Manipulation Program**, available at `http://www.gimp.org/`) in under half an hour apiece.

How are we going to do it?

This project is quite short, and for a change, some of the steps we're going to review have nothing directly to do with writing code. This is because writing the code for Corona's transitions and visual effects is absurdly easy. The challenge in producing good visual effects in Corona is not coding them, but planning which effects will produce the desired results. We will be covering the following topics in this project:

- Planning the dissolve
- Applying the dissolve
- Planning the blood splatter
- Assembling the splatter layers

What do I need to get started?

You should create a new empty project directory called `Predation`, and copy the contents of the `version 0` folder in the project pack directory. This game is functional, and you can try running it before we start modifying it.

Familiarizing yourself with something before you start changing it is always recommended.

Planning the dissolve

Having zombies that blink instantly out of existence when killed is functional for gameplay, but lacks what some developers call the "**wow! factor**". We'd like the zombies to sort of frothily disappear from the side where they come into contact with the diseased goo. While the actual code is simple, the process depends on having the right art assets and placing them correctly, so we need to make sure we understand the plan.

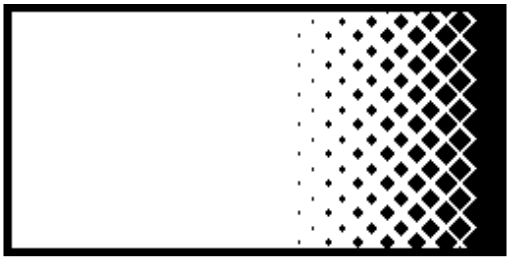
Getting ready

We can't actually edit the sprite, but Corona does offer a feature called masking. **Masking** allows us to use another grayscale image to control which parts of a display object actually appear. Pixels that are black in the mask image are blocked out from the display object on which the mask is used, while white pixels in the mask allow the masked object to show through normally. Mask pixels which are gray allow the masked object to show through with some transparency, according to how dark they are.

We've used simple rectangular masks in previous projects to prevent parts of a scene from bleeding during transitions; in this project we'll practice using them as effects and not just for clipping.

Getting on with it

The real power here comes from the fact that Corona will also let us rotate, slide, or stretch the image being used as a mask, and continue changing these factors over time. We can create a mask image which contains a gradient that blocks progressively more of the image as it goes from one end to the other; and slide that mask across the image to phase it out.

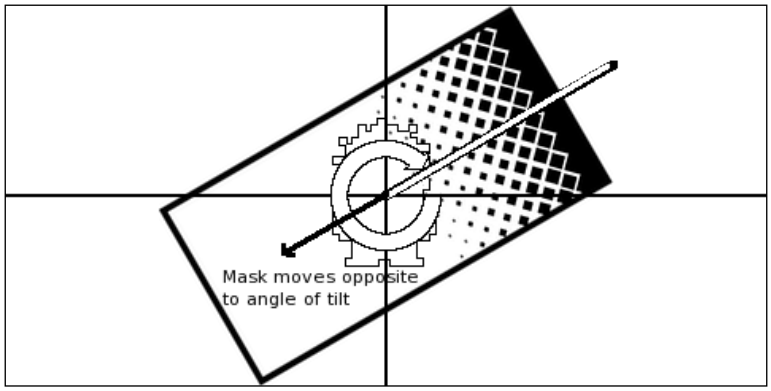


A mask starts out centered over the origin of its target; the center of the image or vector, or the (0,0) point of the group. Its center can be moved to any other coordinate in the target object's system by setting the object's `maskX` and `maskY` properties. The mask can be turned around this point by setting the target's `maskRotation` property; like the rotation property, this value is measured in degrees clockwise. The mask can also be stretched or compressed using the target's `maskScaleX` and `maskScaleY` properties.

If the mask isn't as large as its target, or if the mask is slid so that part of the target is outside it, portions of the target that are outside the bounds of the mask are clipped (not drawn), as if the mask were black there.

To cause the dissolve effect on destroyed zombies to appear to move away from the side facing the cause of its destruction, we can turn the mask so that the increasing black is pointed towards that side. We then move the mask's center away from the destroying object. To keep the motion smooth at any angle, we'll use trigonometry to place the mask's center at a specific radial distance; that is, the distance in a straight line rather than separated into x and y parts.

Keeping in mind that the mask is moving away from the thing it's oriented toward, we'll express the displacement (the distance it moves) as a negative value as shown in the following figure:



What did we do?

By now, we've understood how the existing art assets can be applied to create new effects, and planned out the basics of the math that will create this effect. We're ready to convert the motion and effect that we want into code.

Applying the dissolve

Now that we have the right assets and a plan to use them for the desired effect, it's time to write the actual code.

Getting ready

Open the file `creep.lua` and locate the gap just above the body of the module function, after the table of sprite sequences.

Getting on with it

The first thing that needs to be done is to load the mask file so that it can be used on the zombie sprites.

1. One mask object can be shared among any number of objects, so we'll load it into a variable at the top level of the file:

```
    start = 12, count = 1;
  },
}

local dissolve = graphics.newMask("effects/dissolve.png")

return function(goal, x, y)
```

2. While we're here, we'll add a constant declaration for the distance the mask will travel to move it completely across the sprite until it's invisible. As we discussed in the *Planning the dissolve* section, this distance will be negative since it's moving away from the direction it's turned towards:

```
local dissolve = graphics.newMask("effects/dissolve.png")
local dissolveDisplacement = -72

return function(goal, x, y)
```

3. The mask is now ready and available to the function that needs it; the `self:Die` function inside the constructor, which causes a creep that's touched a hazard to remove itself, leave a pool of hazard, and notify the world. First we'll add the mask to the object that's about to be destroyed:

```
  function self:Die(cause)
    blast(world, cause, self.x, self.y)
    world:dispatchEvent{name='Death'; unit = self, source = cause,
worth = 10, chain = cause.Chain}
```

```
    world:removeEventListener('clock', self)
    self:setMask(dissolve)
    clear(self)
  end
```

4. The mask's grain (the size and spacing of the black areas) is a little coarse for these fairly small images, so we'll scale it down to make it seem finer:

```
    self:setMask(dissolve)
    self.maskScaleX, self.maskScaleY = 0.5, 0.5
    clear(self)
```

For a commercial project, we'd probably want to scale the mask image itself down before release, to produce a cleaner result, but often when a project is in early development, it's more useful to get it running quickly with the assets already at hand, to see results quickly and make changes easily.

5. Next, determine the angle between the dying zombie and the object that triggered its death and was passed into the `Die` function as the cause argument:

```
    self:setMask(dissolve)
    self.maskScaleX, self.maskScaleY = 0.5, 0.5
    local theta = math.atan2(cause.y - self.y, cause.x - self.x)
    clear(self)
```

6. Turn the mask to face this direction. Remember that Lua's trigonometric functions return results in radians, but Corona's rotation properties expect values in degrees.

```
    local theta = math.atan2(cause.y - self.y, cause.x - self.x)
    self.maskRotation = math.deg(theta) % 360
    clear(self)
```

7. Next, animate the mask's position to move a negative amount in that direction. The transition library will animate any numerical property of value of the specified object, so it can control `maskX` and `maskY` just as easily as it can the object's own position or scale.

```
    self.maskRotation = math.deg(theta) % 360
    transition.to(self,
      {
        maskX = dissolveDisplacement * math.cos(theta),
        maskY = dissolveDisplacement * math.sin(theta);
    )
    display.clear(self)
```

8. Finally, instead of clearing the object instantly on death, we want to hold off until the animation is over:

```
    transition.to(self,
      {
        maskX = dissolveDisplacement * math.cos(theta),
        maskY = dissolveDisplacement * math.sin(theta);
        onComplete = display.clear}
    )
  end
```

You can now test the project on either the simulator or a device and watch the zombies evaporate when a splat touches them!

What did we do?

We created a reusable mask that can be attached to each monster object as it dies, and pointed it in the correct direction to appear to come from the direction of whatever killed the creature. We used the transition library to easily manage the movement of the mask that creates the impression of disappearing.

What else do I need to know?

The `display.clear` function isn't a standard part of the Corona `display` module. It's effectively a version of `display.remove` that includes sanity checks as to whether an object is still usable, mostly in cases where the object has been cleared from the scene before its transition finishes. Since it will be used to remove both creeps and blasts, it's in a module of its own, similar to the `math.pythagorean` function in earlier projects.

Of course, you can also animate the other properties of an object that govern its mask, such as its rotation and scale.

Planning the splatter

A more distinctive splatter effect would create some more interest, particularly with the more shooter-oriented, thrill-seeking crowd. This still won't be terribly gory or anything, but we'll give it a more distinctive outline, a little texture, and some variation from one splat to another.

Getting on with it

We want to avoid some of the common traps in effects like this; specifically, uniformity can get distracting or irritating. However, we don't want to create a lot of different animations and pictures.

Rotation is very useful here. We can rotate our splat mask to vary the outline a little; and we can vary the red texture's rotation to similarly mitigate the obvious similarity. Both of these can simply be set randomly from 1 to 360.

While it's not tremendously realistic, we can add a rotation animation to the texture to give it a feeling of motion, and it's very inexpensive. But we still want some steady highlighting, to give an impression of light on liquid, so we can use a faint texture that will continue to point in a single direction. We can make a gleam that brightens the red color while still showing the moving, swirling texture by using a feature called **blend modes**.

A blend mode decides how the object affected changes the image that appears underneath it. The **normal** mode works pretty much as you would expect, simply drawing the object in question over the rest of the image, only leaving gaps for transparent parts of the image, or areas around the border of a shape such as a circle.

The multiply and screen modes work much like they do in Photoshop or similar programs. In **multiply** mode, a black pixel in the multiplied object always leaves black, and a white pixel leaves the original showing through; in **screen** mode, a black pixel leaves the original image and a white pixel gives white. Pixels in between are combined accordingly.

The **add** mode causes the image being overlaid to make the original image selectively brighter; it's similar to screen mode in many ways, but it tends to create a sort of burned in look. We'll combine the add mode with a fairly dim highlight image to add that gleaming look.

The following image illustrates how the same sprite can be combined with a background in each of these modes from left to right: normal, multiply, screen, and add:

What did we do?

We've planned out a slightly more complex effect, one layered together out of several pieces, and included separate animations and effects to bring it together.

Assembling the splatter layers

Now that we have some notion of how to combine the elements, we'll put them together in the actual code. Because we have two different images that we'll want layered together, the splatter object can't be just a single display object any more. We'll replace it with a group, layer on highlights, and add the impression of motion.

Getting ready

Make sure that the effects folder in your project contains the images `blood.png`, `splat.png`, and `highlight.png`. Copy these from the `version 1` subfolder of the `Predation` directory in the project pack if needed.

Getting on with it

Start by opening the file `blast.lua` and perform the following steps:

1. Near the beginning of the file, load the mask:

```
local category = require "category"

local clip = graphics.newMask("effects/splat.png")

local function shrink(object)
```

2. Next, locate the first lines of the main function where a circle is created to represent the splatter and set to appear as red. Change these functions to create a new group parented to the specified world, positioned at a specified point:

```
return function (world, cause, x, y)
  local self = display.newGroup()
  world:insert(self)
  self.x, self.y = x, y

  self:toBack()
```

3. Like the mask image in the zombie example, the blood image isn't currently set to the same size we were using before, so we'll adjust the scale:

```
  world:insert(self)
  self.x, self.y = x, y
  self:scale(25/32, 25/32)

  self:toBack()
```

4. Next, we'll load the blood texture and center it over the group's origin. This keeps the positioning consistent with that of a single image:

```
self:scale(25/32, 25/32)
self.Blood = display.newImage(self, "effects/blood.png")
self.Blood.x, self.Blood.y = 0, 0

self:toBack()
```

5. We'll load the highlight image into the same group, and make sure it's positioned like the blood texture:

```
self.Blood = display.newImage(self, "effects/blood.png")
self.Blood.x, self.Blood.y = 0, 0
self.Blood.rotation = math.random(360)
self.Gleam = display.newImage(self, "effects/highlight.png")
self.Gleam.x, self.Gleam.y = 0, 0

self:toBack()
```

6. We'll set the highlight to `add` mode:

```
self.Gleam = display.newImage(self, "effects/highlight.png")
self.Gleam.x, self.Gleam.y = 0, 0
self.Gleam.blendmode = 'add'

self:toBack()
```

7. Next, we'll attach the mask. Like the zombie dissolve mask, a single loaded mask can be used for all these objects:

```
self.Gleam.blendMode = 'add'
self:setMask(clip)

self:toBack()
```

8. We're almost done. To create that variation we discussed in the previous section, rotate the mask and the texture to random positions:

```
self.Blood.x, self.Blood.y = 0, 0
self.Blood.rotation = math.random(360)
self.Gleam = display.newImage(self, "effects/highlight.png")
self.Gleam.x, self.Gleam.y = 0, 0
self.Gleam.blendMode = 'add'
self:setMask(clip)
self.maskRotation = math.random(360)

self:toBack()
```

9. Finally, we'll animate the blood texture. We'll give it a random turning speed in either direction; the doubling code makes sure that it has a minimum intensity regardless of the direction it's turning in.

```
self.Blood.rotation = math.random(360)
local twist = math.random(-270, 270)
if math.abs(twist) <= 135 then
  twist = twist * 2
end
self.Gleam = display.newImage(self, "effects/highlight.png")
```

10. To finish the animation code, we just need to actually start the animation:

```
if math.abs(twist) <= 135 then
  twist = twist * 2
end
transition.to(self.Blood, {time = 2000; delta = true; rotation =
twist})
self.Gleam = display.newImage(self, "effects/highlight.png")
```

Notice that we don't need to change any of the other code that controls the effect, such as the collision code or the growing and shrinking. It all just works with the new object!

What did we do?

We assembled two layers into a single effect image, and applied a single mask to the whole assembly. We added animation selectively to some parts of the image but not others:

What else do I need to know?

This ability to add a mask to an entire group is one of the most powerful aspects of Corona's masking feature. You can use it for everything from clipping different sections of a display, to creating special effects, to managing important game elements like a flashlight or torch radius effect when dungeon crawling.

Game over – wrapping it up

We've combined a fairly simple set of graphics operations with a set of appropriate art assets to make a working game more visually interesting. Visual appeal is a major part of a game's appeal on any platform and a large part of what will set an independent game apart from other games created by hobbyists. If you're using Corona as a tool for studio development (and even some professional studios like Electronic Arts are beginning to use Corona for mobile development) then techniques like these will make it easier to create the sorts of visuals expected of professional developers on a device of limited power using a straightforward framework.

If you're planning development for a Corona project, keep in mind that good visual effects are likely to require a little more work from artists, and a little less from programmers, because of the emphasis on creating masks and other images to support effects, rather than using code to dynamically create moving images and effects.

Can you take the HEAT? The Hotshot Challenge

Use the remaining art assets to create a crosshair that lingers after each touch that creates a splatter. Experiment with the different image layers available and the `blendMode` property of the display objects: normal, add, multiply, and screen. See what different effects you can create, and different ways to make the crosshairs fade out as time passes.

Project 7

Caves of Glory – Mastering Maps and Zones

The games and projects we've focused on so far have been of an arcade style; gameplay is continuous, time-oriented, and takes place in a single screen. However, many adventure and platform games involve moving the action from one screen to another. For this project, we'll work on reusing the same set of code between several different rooms which must all be visited in the course of a single game.

What do we build?

Caves of Glory will be a treasure-hunting adventure game, a simplified graphical cousin of games like *Colossal Cave Adventure*. The player will direct a character to move through three different maze-like levels until he/she finds each of the eight treasure chests scattered through the levels. The engine we're building is suitable for both scrolling maps and bounded maps; we'll feature transitions between screens, but those screens can scroll.

What does it do?

This project will use a scene template, a modified form of the scene files we've used in previous projects, to create multiple scenes that use the same underlying code, but different level map data. It will manipulate the Lua package loading system so that Corona will load these scenes just as if they were separate scene files. It will use a common set of object and character code to make interactions between the player and the different game elements easily customizable.

The project starts you off with a block of pre-existing code, much of which will look familiar from previous projects. These existing modules supply features like displaying a character, processing input into game commands, and monitoring game logic at the highest level.

One significant difference between this project and previous game projects is that a single game is played out over several scenes; this means that the scene object is no longer a suitable parallel for the game abstraction, the way we've used it up until now. Instead, the game will be an independent object, which can be tracked across multiple scene transitions. Because we still want to be able to monitor the game object for events, we'll represent it with an empty display group parented to the stage. In this way, it will automatically inherit the `addEventListener`, `removeEventListener`, and `dispatchEvent` methods that we've come to rely on so heavily.

Why is it great?

In early versions of Corona, if you wanted to load all files of a particular kind, like map data or sound files, you had to keep an index of what files would be loaded. If you added a new file, it wouldn't appear until you also added it to the index; if you removed one without clearing it from the index, it will typically cause the app to crash when it tries to load a file that doesn't exist.

However, the current version of Corona incorporates the **Lua File System** (**LFS**) library, which allows you to carry out filesystem manipulations that aren't available in base Lua. While changing file permissions is not usually useful in mobile environments, LFS also allows you to create and remove subdirectories in code, which can be useful in conjunction with downloading remote files, and most importantly, it allows you to scan through all the entries in a particular folder. This means that you can do things like build a table view containing a list of all files in a directory, or give each map file its own scene to display that file.

How are we going to do it?

While a few of the components are already built in this example, we're going to focus on the following portions that center around building and displaying the map data.

- Parsing a level file
- Displaying objects
- Creating an efficient background
- Scrolling around a large level
- Interacting with objects
- Defining a chapter
- Creating scenes for datafiles
- Linking scenes together

What do I need to get started?

As usual, you'll need to create your project folder, `CavesOfGlory`, and copy the contents of the `version 0` folder from the project pack directory into it.

Parsing a level file

The first thing we'll do for this project is work on the code that reads a level datafile and build an internal representation out of it; not actual sprites or images, but simply a table describing the various terrain elements, distinctive features, and interactive elements that the level area contains.

Getting ready

The level file format consists of two sections, separated by a blank line. The first section starts with a line that describes the type of terrain that will be used to represent the map: desert, cave, forest, mountain, and so on. The rest of the section is a sort of text picture of the level, rather like the maps used for the *Sokoban* levels in *Project 2*, *SuperCargo – Using Events to Track Game Progress*.

After the blank line is a list of items and objects that appear in the level; a name, possibly preceded by the name of an item-set module that specifies where the item's description is found, the position of a square or range of squares where the item will exist in the level, and any other details about the object, such as what a treasure chest contains or which level an exit leads to. A line might optionally consist of just an item-set name followed by a colon, which sets the default for any following item names that don't specify their own source set.

There are also third-party tools that allow you to draw maps using a graphical interface, such as **Tiled**, which typically stores maps in XML or JSON files; libraries like **Lime** do some of the heavy lifting to load these files into your game. This method has the advantage that it makes it easy for designers to create maps visually; however, the maps aren't usually very human-readable and take up more disk space, as well as sometimes requiring more memory to process.

Getting on with it

Start by opening the file `map.lua` from the project folder. This file provides a single-function module that currently returns an empty table. There are two other functions already in the file, which we'll use to go through the two distinct sections of the file; the small one simply checks a line and returns if it's blank, and the more complicated one takes an iterator, such as we'd usually use to run a single `for` loop, and splits it into two functions, each of which runs its own `for` loop, first up to the first entry that meets the test, then the rest of the way through the iterator. We'll use these functions together to run one loop up to the first blank line (the map image) and one to process everything else in the file (object descriptions).

Custom iterators are one of the most powerful constructions in Lua, but sadly not one of the most intuitive. We'll cover their power, as well as their construction, in greater detail in *Project 9, Into the Woods – Computer Navigation of Environments*.

Splitting the level into map and object data

First, obtain the two iterators that will process the whole file in two sections, separated by a blank line:

```
local self = {}
local layout, objects = split(blank, io.lines(filename))
return self
```

Next, frame in the two loops that will use these sequences:

```
local layout, objects = split(blank, io.lines(filename))
for map_line in layout() do
end
for object in objects() do
end
return self
```

We'll fill in the first loop with some basic structure code to prepare for processing the second loop.

Creating the map canvas

In order to add objects to the map, we need to have the representation of those spaces stored in the map array, so we'll go over each line and character in the map image and add a corresponding space to the map.

Skipping the terrain description, add a row to the map for each line in the first part:

```
for map_line in layout() do
  local terrain = map_line:match('^(.+):$')
  if terrain then
  else
    local row = {}
    table.insert(self, row)
  end
end
```

For each character in the map line, add a space to the row. Each space will have additional data added to it when we develop the map portion more fully:

```
    local row = {}
    table.insert(self, row)
    for tile in map_line:gmatch('.') do
      local space = {
        Features = {},
      }
      table.insert(row, space)
    end
  end
```

Reading objects into the position data

Each line will either be an object including a set prefix, a set prefix alone to be used for later objects without one, or an object without any set prefix that uses the last declared prefix. So before the loop starts, we'll need to stake out a variable to store the last declared default prefix:

```
end
local family_group
for object in objects() do
```

So, the first thing we need to do is check whether the line meets the description of a new default prefix; a word followed by a colon:

```
for object in objects() do
  local family = objectName:match("^(%S+):$")
  if family then
    family_group = family
  end
end
```

Otherwise, the object needs to be processed according to its identifier and position (and any other specifics):

```
if family then
  family_group = family
else
  local objectName, data = object:match("^(%S+)%s*(.*)$")
end
```

If the object name doesn't contain its own set prefix, we need to apply the current default set prefix:

```
else
  local objectName, data = object:match("^(%S+)%s*(.*)$")
  local family, name = objectName:match("^([^:]+):?(.*)$")
  if name == '' then
    family, name = family_group, family
  end
end
```

Each object in the list has a position on the map, and some objects have more info; for example, an exit to another level should specify which level it connects to and where in the destination level it takes the player. We'll separate this optional data from the position info and split the position into its x and y parts, separated by a comma:

```
  if name == '' then
    family, name = family_group, family
  end
  local x, y, details = data:match("^([-%d]+),([-%d]+)%s*(.*)$")
end
```

Each part is able to contain a range to support large objects, so we'll process the x and y portions to check for a dash or multiple numbers as needed:

```
  local x, y, details = data:match("^([-%d]+),([-%d]+)%s*(.*)$")
  local xMin, range, xMax = x:match('(%d*)(%-?)(%d*)')
  xMin = tonumber(xMin) or 1
  xMax = tonumber(xMax) or range ~= '-' and tonumber(xMin) or
#self[1]
  local yMin, range, yMax = y:match('(%d*)(%-?)(%d*)')
  yMin = tonumber(yMin) or 1
  yMax = tonumber(yMax) or range ~= '-' and tonumber(yMin) or
#self
end
```

Finally, we'll generate the specified object and add it to each of the spaces in the specified range. Each set prefix will define a module in the objects package, containing one constructor function for each object in the set:

```
        yMax = tonumber(yMax) or range ~= '-' and tonumber(yMin) or
#self
        for x = xMin, xMax do
          for y = yMin, yMax do
            table.insert(self[y][x].Features, require("objects."..
family)[name](details))
          end
        end
      end
```

Save `map.lua`.

What did we do?

We skipped over the first section of a two-part file (for the most part), read each entry in the second section, and tracked a changeable setting across multiple entries. We stored instances of the specified objects in the locations they were tagged as belonging in.

Displaying objects

Now that we have a list of objects and positions, we'll add representations of the objects in a map to the scene created from that map. Because a map is only loaded once in the lifecycle of a scene, but the visuals might be purged to conserve memory, we don't simply attach object data to the sprites or images used to display the objects, as we did in simpler projects.

Getting on with it

Inside your project directory, find the `objects` folder that you copied over when you started the project, and open the file `common.lua` in that folder (if you've looked at the map datafiles, you'll notice this file has the same name as the object set used in those files). Notice that this file already loads an image sheet and uses an internal table of sequence lists for sprites.

Supplying visual descriptions of items

Find the table constructor assigned to `common.campfire` and add a new function definition to this table:

```
common.campfire = object {
  Embody = function(self)
  end;
  EnterSpace = true;
}
```

This function creates and loads a campfire sprite from the preloaded sheet, which contains several different objects and frames, and defined sequences that specify which frames make up the campfire animations:

```
  Embody = function(self)
    return display.newSprite(sheet, sequences.campfire)
  end;
```

In the constructor for the `common.barrel` table, add an `Embody` function that creates a static image from a frame of the same sheet:

```
common.barrel = object {
  Embody = function(self)
    return display.newImage(sheet, 22)
  end;
  EnterSpace = false;
}
```

Storing multiple static images in a single texture in this way is often called **texture atlasing**. It offers a way to consolidate the use of texture memory and improves performance in many circumstances.

The `Embody` function for treasure chests will be similar to the campfire, so that it can be switched between open and closed visual states:

```
common.chest = object {
  Embody = function(self)
    return display.newSprite(sheet, sequences.chest)
  end;
  EnterSpace = false;
}
```

The `exit` object won't have an `Embody` function, because it doesn't have its own appearance; it simply designates an area of the map as being a trigger. The cave that makes this area obvious to the player will be part of the background.

Loading item visuals into the world

Now that the object templates can specify their appearances, we need to generate those objects and arrange them in the scene. Save any other files you have open, and open the `world.lua` file. This will be the module that scenes use to build the groups that display the contents of their maps.

First, create a subgroup to hold objects, characters, and other local features:

```
local self = display.newGroup()
self.Features = display.newGroup()
self:insert(self.Features)
function self:View(x, y)
```

Next, add a function to the world group, which will add a specified object to the world's map at the point you declare:

```
end
function self:Add(object, x, y)
  local list = self.Map[y][x].Features
  if not table.indexOf(list, object) then table.insert(list, object)
  end
end
return self
```

If the object is already represented in the world, simply return that representation. Otherwise, be ready to add one if needed.

```
if not table.indexOf(list, object) then table.insert(list, object) end
if self[object] then return self[object] end
 if object.Embody then
 end
end
```

This function will call the object's `Embody` function as we declared it before and place it within the space of the chosen tile:

```
  if object.Embody then
    local actor = object:Embody()
    self.Features:insert(actor)
    self:Place(actor, x, y)
    return actor
  end
```

Now, add a new function to the new world object, which will load it with the contents of a map created using `map.lua`:

```
end
function self:Load(map)
end
return self
```

This function will go through each space in the map:

```
function self:Load(map)
  for v, row in ipairs(map) do
    for h, space in ipairs(row) do
    end
  end
end
```

It will scan that space for objects, and use the function we declared earlier to create each one within the world, in the center of its tile:

```
    for h, space in ipairs(row) do
      for i, feature in ipairs(space.Features) do
        self:Add(feature, h, v)
      end
    end
```

Finally, attach the `map` file to the world for future reference:

```
function self:Load(map)
  self.Map = map
  for v, row in ipairs(map) do
```

Save the file. This should be the first point at which you can load the project in the simulator, walk your character around by tapping the various edges of the screen, and see a handful of objects move around you.

The next step will be filling in the background, so that you're not moving among these objects in a featureless void.

What did we do?

We used per-object functions to map image and sprite data to objects in a generalized fashion. We looked at each map space in turn and loaded all the relevant objects for that space in their specified order.

What else do I need to know?

One thing you may want to note is the way the sheet size is specified in the file as being 224 pixels squared, when the actual image is 112 pixels squared. In a case like this where you want to scale up all of the contents (the icons are 16 x 16 but are being used on a map of 32 x 32 tiles), this is a fast and simple way to do it.

Another approach that can be useful in conjunction with sprite sheets and texture atlases is to employ a texture-packing utility. When a texture is loaded into memory, each of its dimensions is increased if needed to make it a power of two; so while a 60 x 120 image will take up memory as if it were 64 x 128, a 550 x 550 image will take up the same texture memory as a 1024 x 1024 background texture! Sometimes tiled textures can be rearranged with a different number of tiles per side, which allows them to reduce one dimension below the next power of two without pushing the other dimension up over its next power of two.

More commonly, however, texture packers help by taking sprites and similar contents that don't take up their entire tile space and squeezing them together to reduce the total area needed for all sprites in the sheet. These utilities can typically export a Lua file that contains all the tables to describe the locations of the different frames, along with their rearranged image file.

Creating an efficient background

While objects and characters will typically be scattered around a level, covering only a small fraction of its total area, the ground generally takes up the whole area of the level. This isn't a problem on small levels that are mostly or completely contained within the screen, but in large, scrolling levels, the number of objects off-screen can begin to have a significant effect on performance.

Getting on with it

To display the background terrain of a level, first we're going to need to convert the one-character tiles from the level map file into sequences appropriate for the terrain type as the level loads. The core of the conversion will be handled by the terrain type itself.

Loading background data from a level file

Save any open files and reopen `map.lua`. When each line is tested to see if it is a terrain specifier, add code to save that choice with the map:

```
local terrain = map_line:match('^(.+):$')
if terrain then
  self.Terrain = require("terrain."..terrain)
else
```

However, if a terrain specifier tries to replace another one, or if terrain content is loaded before any specifier is found, an error is thrown:

```
    if terrain then
      assert(not self.Terrain, "Error in map "..filename..": multiple
terrain types specified")
      self.Terrain = require("terrain."..terrain)
    else
      assert(self.Terrain, "Error in map "..filename..": No terrain
type specified for map content")
      local row = {}
```

Obviously, you don't want your game to ship with faulty maps in it. But one of the reasons for supporting a file format like this, as in *Project 5*, *Atmosfall – Managing Game Progress with Coroutines*, is to facilitate other designers or scripters working on the game who may not be full-time programmers, and who will usually appreciate error messages that tell them why their file is broken. For that matter, you may make a mistake yourself, and be glad for the clue in tracking it down.

In the body of the loop that expands each line of terrain data, expand the fields it stores with each space:

```
        local space = {
          map = self;
          x = #row + 1,
          y = #self;
          Features = {};
        }
```

For each character in the file, we will also request that the terrain expand it to the appropriate animation sequence. So, if maps typically use the # character to represent a wall or obstacle, a desert terrain might expand it to rock while a forest terrain might expand it to bush.

```
        y = #self;
        Ground = self.Terrain:Expand(tile);
        Features = {};
```

Finally, after assembling the individual tiles, we ask the terrain type to clean up the map. For instance, it might adjust border tiles so that they use the correct edge or corner based on which side of a pool or hole or other feature they are located on.

```
  end
  if self.Terrain then
    self.Terrain:Polish(self)
  end
```

```
local family_group
for object in objects() do
```

Save `map.lua`.

Assembling the tile field

Open `world.lua`. At the beginning of the module function, before creating the features layer, add a new layer that will contain the terrain tiles. Since every tile in this layer will be a sprite from the same sheet, we can improve performance by using Corona's `newImageGroup` function:

```
local self = display.newGroup()
self.Ground = display.newImageGroup(terrain.Atlas)
self:insert(self.Ground)
self.Features = display.newGroup()
```

Request the size of a single tile from the terrain type, so that tiles can be fit together properly:

```
self:insert(self.Ground)
self.HSize, self.VSize = terrain:TileDimensions()
self.Features = display.newGroup()
```

Create an array (which will eventually be two-dimensional) to store the tiles we create:

```
self.HSize, self.VSize = terrain:TileDimensions()
self.Tiles = {}
self.Features = display.newGroup()
```

Next, add tiles to the layer to cover the screen. Note that we add an extra layer of tiles around the bottom and right. This means that we don't have gaps showing during the times the background is being aligned with the screen as it moves, in the next section.

```
self.Tiles = {}
for v = 1, rows + 1 do
  local row = {}
  self.Tiles[v] = row
  for h = 1, columns + 1 do
    local tile = terrain:Tile()
    self.Ground:insert(tile)
    tile:setReferencePoint(display.BottomRightReferencePoint)
    tile.x, tile.y = h * self.HSize, v * self.VSize
    row[h] = tile
  end
end
self.Features = display.newGroup()
```

What did we do?

We've created a layer of terrain segments large enough to cover the screen. These segments can be configured to display any section of the background that fits within the confines of the screen.

What else do I need to know?

Performance concerns aren't quite as large as they used to be in earlier versions of Corona. Around the same time that image sheets and image groups were introduced, Corona instituted a feature called **offscreen culling**, wherein it doesn't spend time drawing objects that are not currently visible. However, while this makes the processing time for offscreen tiles very small, it doesn't reduce it to zero, so a performance impact might still be noticeable for very large worlds if you generated enough tiles to hold the whole thing.

Moreover, loading times are still a challenge for mobile games, since phones and tablets have comparatively limited memory. Using this window onto the terrain means that noticeably less time needs to be used to construct the world object during this easily overloaded program phase.

Regarding the `terrain:Polish()` function, it constitutes a large portion of the work done to make a level look good. However, it's also not very interesting; it consists mostly of a long chain of `if...then...else...if` statements. For this reason it's not being discussed here in detail; you can review the individual biome files in `terrain.lua` if you're curious about what it does.

Scrolling around a large level

The result of reducing the background to a screen-sized window is that it has to be kept placed behind whatever section of the world is currently visible in that screen.

Getting on with it

Open `world.lua`, if needed.

Displaying the visible background

Insert a new local function before the module function, which will set the world tiles to display the needed portion of the background:

```
local function alignBackground(world, ground)
end

return function (terrain, columns, rows)
```

First, determine which tile appears in the top-left corner of the screen:

```
local function alignBackground(world, ground)
  local x, y = world:contentToLocal(0, 0)
  x, y = math.floor(x / world.HSize), math.floor(y / world.VSize)
end
```

Next, go through each of the tiles of the visible background:

```
  x, y = math.floor(x / world.HSize), math.floor(y / world.VSize)
  for v, row in ipairs(world.Tiles) do
    local sourceRow = world.Map[y + v]
    for h, tile in ipairs(row) do
    end
  end
end
```

For each one, set it to display the appropriate piece of terrain, based on which tile of the original map would be in that section:

```
    local sourceRow = world.Map[y + v]
    for h, tile in ipairs(row) do
      local space = sourceRow and sourceRow[x + h]
      if space then
        tile.isVisible = true
        tile:setSequence(space.GroundType)
        tile:play()
      end
    end
  end
```

If the tile has slipped off the bounds of the map, hide the tile until the map moves back:

```
      if space then
        tile.isVisible = true
        tile:setSequence(space.GroundType)
        tile:play()
      else
        tile.isVisible = false
      end
```

Aligning the background with the screen

This keeps the background showing the part of the world that should be on screen, but the background object itself doesn't move within the world as the world scrolls. This, however, can be fixed easily:

```
          tile.isVisible = false
        end
      end
    end
    ground.x, ground.y = x * world.HSize, y * world.VSize
  end
```

The background snaps to tile boundaries as it is moved about.

Finally, make sure that the function actually gets called whenever the map is updated:

```
    function self:View(x, y)
      matchPointPlacement(self, x, y, display.getCurrentStage(),
  display.contentCenterX, display.contentCenterY)
      alignBackground(self, self.Ground)
    end
```

What did we do?

We created a function that links the visible portions of a world to the slice of the map structure that describes them, and positions the tiles so that they stay in sync with the world as it is being moved.

Interacting with objects

The game would be singularly uninteresting if all you could do is walk around the levels and look at them.

Getting on with it

Open `world.lua`, if needed, and find the function `self:Add`. Our first step here will be to register the world to detect taps on those objects that the player can manipulate.

Registering interactive objects with the world

Check whether the new object has an `Interact` function. If it does, add the world as a tap listener on the new object:

```
      self:Place(actor, x, y)
      if object.Interact then
```

```
    actor:addEventListener('tap', self)
  end
  return actor
```

Next, add a tap handler to the world object:

```
    alignBackground(self, self.Ground)
end
function self:tap(event)
end
function self:Add(object, x, y)
```

Because the player character represents the player in the game world, a game object can only be interacted with if the player character is next to it. The first step to determine this is to identify the tile the object is on from the event coordinates:

```
function self:tap(event)
  local x, y = self:contentToLocal(event.x, event.y)
  x, y = math.ceil(x / self.HSize), math.ceil(y / self.VSize)
end
```

Next, the nearby squares are checked for the presence of at least one character (we'll write this function momentarily):

```
    x, y = math.ceil(x / self.HSize), math.ceil(y / self.VSize)
    local character = seekAdjacentCharacer(self.Map, x, y)
    if character then
    end
end
```

The object being interacted with is passed the specific character who is using it, in case it has some special effect on them:

```
    if character then
      event.target.Info:Interact(character)
    end
```

At the top level of the file, add the function to scan adjacent squares for a character:

```
  ground.x, ground.y = x * world.HSize, y * world.VSize
end

local function seekAdjacentCharacter(map, x, y)
end

return function (terrain, columns, rows)
```

This function will test adjacent offsets from the space where the object is located, starting with the space above it, until it reaches that space again:

```
local function seekAdjacentCharacter(map, x, y)
  local dX, dY = 0, -1
  repeat
  until dY == -1
end
```

To advance between clockwise adjacent spaces, the loop will use a programmers' trick to swap the x and y offsets, negating the y offset to ensure that it covers all four spaces:

```
  repeat
    dX, dY = -dY, dX
  until dY == -1
```

In each space, the function will look through the local features and see if any of them is a character. If so, its work is done and it returns what it has found.

```
  repeat
    local row = self.Map[y + dY]
    local space = row and row[x + dX]
    if space then
      for _, feature in pairs(space.Features) do
        if feature.Character then
          return feature
        end
      end
    end
    dX, dY = -dY, dX
  until dY == -1
```

Adding interactivity to an object definition

Finally, we need to add a description, wherever appropriate, to objects that are interactive, so that the world knows to monitor them. The common set contains only one interactive object, the chest. Add the `Interact` function to the `common.chest` definition in `objects/common.lua`:

```
common.chest = object {
  Embody = function(self)
    return display.newSprite(sheet, sequences.chest)
  end;
  EnterSpace = false;
  Interact = function (self, player)
  end;
}
```

This function will be responsible for using a function on the interacting player to give them a mark of progress toward their goal, a new item in their inventory (distinct from an item in the world):

```
Interact = function (self, player)
  if not self.Opened then
    self:setSequence("open")
    player:Give("coin")
    self.Opened = true
  end
end;
```

What did we do?

We added conditional touch registration, so that the system will only spend time processing those objects that do something when touched.

Defining a chapter

The goal of this project has been loading content automatically and handling multi-part areas, so it's important to include features like quest definitions and to specify which area in a chapter the player will begin in.

Getting ready

Create a file in the `chapter/1` folder of the project directory, `contents`, and open it in a text editor.

Getting on with it

This chapter is simple and needs only two pieces of information.

Specifying a chapter's beginning and end

On the first line of the chapter content file, specify the chapter start:

```
start:
```

The chapter will begin in the `desert_camp` map, on a square next to the tent decoration:

```
start:  desert_camp  8,6
```

On the next line, we'll detail the chapter's goal: the collection of eight coin objects:

```
start:  desert_camp  8,6
goal:    coin  8
```

Launching a selected chapter

Save the `content` file and open `game.lua`. This module creates the game object and attaches its relevant functions. The `game:Begin()` function is intended to launch a particular chapter. In order to do this, it will read the `content` file to place the character in the right scene and note which accomplishments it needs to track on the player to determine when the chapter is over. Start by looping over the lines of the `content` file for the specified chapter:

```
function self:Begin(chapter)
  local target, x, y
  for line in io.lines(directory.."/contents") do
    local action, details = line:match("^([^:]+):%:%s+(.+)$")
  end
end
```

When a start line is found, record which scene and location it needs to go to; we'll finish processing the file before launching the actual transition:

```
  for line in io.lines(directory.."/contents") do
    local action, details = line:match("^([^:]+):%:%s+(.+)$")
    if action == 'start' then
      target, x, y = details:match("^(%w+)%s+(%d+)%,(%d+)")
      x, y = tonumber(x), tonumber(y)
    end
  end
```

When a goal line is found, record the type of goal being sought and how many are required:

```
    if action == 'start' then
      target, x, y = details:match("^(%w+)%s+(%d+)%,(%d+)")
      x, y = tonumber(x), tonumber(y)
    elseif action == 'goal' then
      local kind, count = details:match("^(%w+)%s+(%d+)")
      self.GoalKind, self.GoalCount = kind, tonumber(count)
    end
```

Tracking goal progress

While we're processing this file, go to the `Proceed` function in the `gotoScene` call further down, and register the game to receive events that indicate the player gained a new item:

```
self.Player = character
    self.Player:addEventListener('Gained', self)
    UI:AttachPlayer(self.Player)
```

Add a function to the game object to respond to user when these events are received:

```
self.isVisible = false
function self:Gained( event )
end
function self:Begin(chapter)
```

This handler will compare the received prize to the desired goal item to determine if it should be counted:

```
function self:Gained( event )
  if event.object == self.GoalKind then
    self.CountAcquired = (self.CountAcquired or 0) + 1
  end
end
```

If the count has reached the needed number of items, the chapter is declared completed:

```
  if event.object == self.GoalKind then
    self.CountAcquired = (self.CountAcquired or 0) + 1
    if self.CountAcquired >= self.GoalCount then
      self:dispatchEvent{name = 'Goal'; action = 'completed', kind =
self.GoalKind}
      self:dispatchEvent{name = 'Chapter'; action = 'completed',
index = self.Chapter}
    end
  end
```

What did we do?

While the separate map files in a chapter folder specify its various zones and goodies, we've added a file that indicates how and where the chapter should begin, and what special deeds are needed to finish it.

Creating scenes for datafiles

The code that loads a map into a scene is meaningless unless scenes are created that will actually load those specific files. Loading a specified chapter into the game will also include building a scene for each datafile.

Getting on with it

At the start of the `game:Begin()` function in `game.lua`, add a loop to examine each file in the specified chapter directory.

Scanning the chapter directory

Add the chapter number onto the reserved directory path, and use the `lfs.dir` iterator to run a `for` loop over each file in the directory:

```
function self:Begin(chapter)
  self.Chapter = chapter
  local directory = chapterDirectory .. chapter
  for name in lfs.dir(directory) do
  end
  local target, x, y
```

We don't want to create scenes for files that aren't maps, like the `content` file:

```
  for name in lfs.dir(directory) do
    if name:match("^[^.]+%.map$") then
    end
  end
```

For each map file, we'll run our scene creation engine with that file to generate a loader for a new scene. This doesn't actually make the scene itself immediately, but it puts it where `storyboard.gotoScene` can find it later:

```
    if name:match("^[^.]+%.map$") then
      scene(name)
    end
```

Launching the first scene

Once all other work required for initiating a chapter is finished, and the starting scene is identified, we need only to launch it from the data we obtained in the chapter content file:

```
    UI:AttachPlayer(self.Player)
    storyboard.gotoScene(target, {params = {Characters = {[self.
Player] = {x = x, y = y} } } })
  end;
```

What did we do?

We used the LFS library to search for any files that have been designated part of a chapter, and used our scene template module to create scene loaders for them.

What else do I need to know?

The key to these preloaded scenes is the `package.preload` table. When you call `require`, it could find the desired module in any of a number of places, including regular files when you run your project in the simulator, archive contents when you run it on a device, and even dynamically-linked libraries if you are using Corona Enterprise. But the first place it looks is the table `package.preload`, where it will run any function it finds under the module name as the chosen module loader, much as if that function were the contents of a file.

Linking scenes together

Now that we have a chapter that contains multiple scenes, the last step is to make it possible to travel between them.

Getting on with it

We need to adjust the function that moves characters and features around the map, so that it can tell when a character has stepped into a trigger region (an exit into another level, in this case). Open the `map.lua` file and find the `self:Move()` function definition.

Recognizing trigger movement

When an item is moved, it returns an event target that will receive a `Moved` event when any animations or other transitions are complete. The calling code can listen to this target to determine when movement is finished and effects can be applied.

```
        table.insert(space.Features, table.remove(origin.Features,
  table.indexOf(origin.Features, item)))
        item:Move(origin, space)
          :addEventListener('Moved', self)
      end
```

The listener needs a handler function that will scan features in the new space:

```
  function self:Moved( event )
    for _, feature in ipairs(event.destination.Features) do
    event.target:removeEventListener('Moved', self)
  end
  function self:Move(item, dX, dY)
```

For each feature identified, it will execute any code attached to that feature for handling an object entering its space:

```
for _, feature in ipairs(event.destination.Features) do
  if feature.SpaceEntered then
    feaure:SpaceEntered(event.target)
  end
end
event.target:removeEventListener('Moved', self)
```

Executing the level transition

Save `map.lua` and open `objects/common.lua`. Add an `Initialize` function to the description for `common.exit` that will process the transition target details from the map file:

```
common.exit = object {
  Initialize = function(self, details)
    local targetScene, targetX, targetY = details:match("^(%w+)%:(%d+
),(%d+)")
    self.Target = {
      scene = targetScene;
      x = tonumber(targetX), y = tonumber(targetY)
    }
  end;
  EnterSpace = true;
}
```

Once this is done, add a `SpaceEntered` function to the same object:

```
  EnterSpace = true;
  SpaceEntered = function (self, incoming)
  end;
}
```

This function will transition to the desired scene, passing the information about where the character should be placed when the scene starts:

```
  SpaceEntered = function (self, incoming)
    if incoming.Character then
      storyboard.gotoScene(self.Target.scene, { params = { Characters
= { [incoming] = {x = self.Target.x, y = self.Target.y} } } })
    end
  end;
```

What did we do?

We added recognition for characters entering specific spaces. Up to this point, we have relied on the physics engine for most such detection, but for tile-based games collision detection becomes trivial.

We also added a response to that feature collision, allowing the player to navigate between all the scenes of the chapter.

Game over – wrapping it up

This project has been an exercise in modularity; loading from datafiles stored in a format that is easy to author and update using simple tools, automatically adding new level scenes based on datafiles as they're added to the project (without having to update any indexes or similar maintenance), creating links between scenes based on name and destination only, and separating out visible terrain backgrounds that fill the screen only from the conceptual maps that describe the whole environment.

While the gameplay is minimal in this example, the techniques developed here can be combined with other development approaches to build involved and complex games.

Can you take the HEAT? The Hotshot Challenge

The game currently has a couple of weaknesses; it's not persistent in the event the user closes it to use something else, and in the event that a previously visited scene gets unloaded, the chests in that room will reset, allowing the player to finish the level without visiting all chests.

Make the game persistent; preserve the statuses of the various chests in a file so that they can be set back to their old statuses when the game is reloaded. Refer back to *Project 2*, *SuperCargo – Using Events to Track Game Progress*, if you need examples of saving game status advances.

Project 8

The Beat Goes On – Integrating with Game Networks

Games with a social aspect have become increasingly popular, and Corona offers a way to access these networks and migrate your app from one to another according to your target platform with a minimum of code changes using the `gameNetwork` library. Along the way, we'll also practice exploiting the facility of most touchscreen devices to detect and track multiple points of touch, allowing for a greater variety of gameplay.

What do we build?

This project presents you with a completed game engine—almost. The game can load and play level files, but doesn't currently accept any player input, so it's impossible to complete the level or score any points.

You will incorporate recognition for multiple touches, expand the game play to evaluate these touches for accuracy, and score the player accordingly. Once the level is over, you will submit the player's new score to a centralized leaderboard for approval.

What does it do?

The project searches for game files that contain descriptions of what targets to offer the player at what times, and uses a table view to list these files to the player to choose from. When the player chooses a game file, the game looks for a music file with a matching name, and plays it if it exists.

During the course of the game, circular targets will appear on the screen, either flashing in and out or drifting around the screen. The player's goal is to touch as close to the centers of these circles as possible, and keep his or her fingers close to the centers as they move. As time passes for each target the player is touching, the player's score will increase; the closer their touches are to the centers of the targets, the faster their score will go up.

The app will use a natively available game network to include players' final scores in a shared leaderboard. This supports only one network, Apple's Game Center, but developers building for Android devices can enhance the project with Corona's new Corona Cloud service.

Corona Cloud had just been made available at the time of this publication and could not be included, but a library to make it compatible with the `gameNetwork` module was in development.

Why is it great?

Gamers are increasingly showing that they prefer playing in a context with other people, where they can be a part of a game community. While Corona currently supports a limited number of game networks, it offers the features of those networks in a way which is intended to be independent of any one operating system. So as new services become available in the future, keeping your projects up to date using your service of choice should remain fairly simple.

How are we going to do it?

Like *Project 3*, *TranslationBuddy – Fast App Development for any Field*, a substantial part of this project will be making sure that the settings for the app are configured properly in a web service, rather than programming the supporting code. We will be covering the following tasks in this chapter:

- Tracking multiple touches
- Comparing touches with targets
- Loading and playing music
- Enabling Game Center on the Provisioning Portal
- Enabling Game Center on iTunes Connect
- Initializing a game network connection
- Posting scores and achievements

What do I need to get started?

You can complete the first portions of this exercise, tracking touches and playing background music, on any device; but the only active game network that the project was written to support is Apple's Game Center. Since Game Center is not compatible with non-Apple devices, to test the second part, you'll need an Apple Developer Account, an iTunes Connect login, and an iOS device capable of running Corona apps (iOS 4.3 or more recent).

You'll also need to create a project folder called `TheBeatGoesOn` and to copy the contents of the `version 0` folder in the project pack directory to it. This includes the base game scene, splash screen, a music file, and a game description file that specifies what targets will appear and how they will move over the course of the song.

Tracking multiple touches

The first thing we will do is to construct a list that contains all currently active touch objects and their locations. It will use its own touch listener to stay up to date with any touch event that reaches the `Runtime` target (so buttons and other objects that return true in their touch listeners will not affect it).

Getting ready

After you've copied your project directory from the `version 0` contents, create a text file at the top level of your project called `touches.lua` and open it.

Getting on with it

This module will allow the user to iterate through all the currently active touches.

Enabling multiple touches

Since this file exists to track multiple simultaneous touches, it should guarantee that Corona will track these touches; by default, Corona operates in single-touch mode.

```
system.activate("multitouch")
```

Testing multi-touch events requires building for device; since the simulator emulates "touch" input based on the position of the mouse, it can't generate multiple touch points at once.

Creating a list

We will need a table to track touches across individual event calls:

```
system.activate("multitouch")

local touchList = {}
```

Adding new entries to the list

We'll need to use a touch listener on the `Runtime` target to follow individual touches:

```
local touchList = {}

local function process( event )
end
Runtime:addEventListener('touch', process)
```

When a touch begins, add its location info to the touches list:

```
local function process( event )
  if event.phase == 'began' then
    touchList[event.id] = {x = event.x, y = event.y; xStart = event.
xStart, yStart = event.yStart}
  end
end
```

Touch events contain an ID field, which contains a value that identifies the touch. This value will be the same for all touch events that are part of the same gesture or finger contact.

Updating entries in the list

When a touch moves around the screen, we make sure that it's one we're tracking (in case something else intercepted the beginning of a touch), and that we update the correct info:

```
    touchList[event.id] = {x = event.x, y = event.y; xStart = event.
xStart, yStart = event.yStart}
  elseif event.phase = 'moved' then
    local touch = touchList[event.id]
    if touch then
    end
  end
```

If we recognize the touch, we update it with the current location:

```
    if touch then
      touch.x, touch.y = event.x, event.y
    end
```

Clearing entries from the list

Touch events that are not moved or begun are either ended or cancelled, either of which means that the touch is over:

```
      touch.x, touch.y = event.x, event.y
    end
  else
    touchList[event.id] = nil
  end
```

Granting access to the list

We won't grant direct access to the list to avoid making it easy for coders relying on the module to damage it, but instead we will pass a function that iterates over the list's contents:

```
Runtime:addEventListener('touch', process)

return function()
  return pairs(touchList)
end
```

What did we do?

This list allows us to check the status of all active touches whenever we need to, by centralizing responses to the touch events rather than trying to process each one as it comes in. This also means that we have access to each touch that hasn't received any events since the last cycle.

Comparing touches with targets

Now that we have a list of touches that can be examined at will, we need to add logic to compare those touches with the desired target locations.

Getting ready

Save `touches.lua` if needed, and open the file `game.lua` from the project folder. Take a look through the `scene:willEnterScene(event)` function; this is where the list of game targets will be created, that we will use in the main loop.

Getting on with it

Start at the top of `game.lua`.

Loading the touches module

Load the function that will carry a `for` loop over all active touch points:

```
local level = require "level"

local touches = require "touches"

local storyboard = require "storyboard"
```

Matching touches to targets

Find the `scene:clock(event)` function which is in charge of updating the game each time its main loop runs; it is currently empty.

1. The first thing it does is start an empty list, which will hold targets that have a touch nearby, and the distance to the nearest touch for those targets, as well as a count of active touches that aren't the closest to any target:

```
function scene:clock( event )
  local orbits = {}
  local orphans = 0
end
```

2. For each touch currently on the screen, it considers each target and finds the nearest one to that touch:

```
function scene:clock( event )
  local orbits = {}
  local orphans = 0
  for id, location in touches() do
    local nearest, proximity
    for _, target in pairs(self.Targets) do
      local distance = math.pythagorean(location.x - target.x,
location.y - target.y)
      if not nearest or distance < proximity then
        nearest, proximity = target, distance
      end
    end
  end
end
```

3. If the touch is the closest touch to some target, and within a specified range, it records the distance:

```
      if not nearest or distance < proximity then
        nearest, proximity = target, distance
      end
    end
```

```
    if proximity <= self.TARGET_RADIUS and ( not orbits[nearest]
or proximity < orbits[nearest]) then
      orbits[nearest] = proximity
    end
  end
```

4. Otherwise, it marks the touch as orphaned, or off-target. This count is stored as a negative number to make it easy to subtract from the user's score later.

```
    if proximity <= self.TARGET_RADIUS and ( not orbits[nearest]
or proximity < orbits[nearest]) then
      orbits[nearest] = proximity
    else
      orphans = orphans - 1
    end
```

Adjusting the player's score

In the same function, we'll adjust the player's score. Because accuracy in this game is something that has to be sustained, the total accuracy of touches at any given moment is used to speed up or slow down the rate at which the score increases. If the score stops increasing completely, the level is over.

1. First, reckon the penalty from any inaccurate touches:

```
      orphans = orphans - 1
    end
  end
  local pressure = orphans * self.TARGET_RADIUS * -0.5
end
```

2. Next, consider each touch that was on-target, and adjust the score increase upward accordingly.

```
  local pressure = orphans * self.TARGET_RADIUS * -0.5
  for target, proximity in pairs(orbits) do
    pressure = pressure + (self.TARGET_RADIUS - proximity)
  end
end
```

3. Apply this rate change to the degree of score increase, adjusted for the actual amount of time that has passed.

```
    pressure = pressure + (self.TARGET_RADIUS - proximity)
  end
  self.Velocity = math.min(self.Velocity + pressure * event.delta,
self.TERMINAL_VELOCITY)
end
```

4. If the rate has bottomed out, the level is over.

```
  self.Velocity = math.min(self.Velocity + pressure * event.
elapsed, self.TERMINAL_VELOCITY)
  if self.Velocity <= 0 then
    self:dispatchEvent{name = 'Level'; action = 'failed'}
  end
end
```

5. Otherwise, adjust the score by the rate of change multiplied by the amount of time passed.

```
  if self.Velocity <= 0 then
    self:dispatchEvent{name = 'Level'; action = 'failed'}
  else
    self.ScoreCurrent = self.ScoreCurrent + self.Velocity * event.
delta
  end
end
```

What did we do?

We set up the main game loop to award points for each touch that lies close enough to a target point (keep in mind that the target points are in motion). Because this will happen in every frame, we need to adjust the value being gained according to the amount of game time that has actually passed.

Loading and playing music

While it is possible to play a rhythm game without any audio, it certainly reduces the experience.

Getting ready

Make sure that you copied the file `RockYourBody|Vibemaster B.mp3` and that it resides in the `levels` folder next to the file `RockYourBody.easy`.

Getting on with it

We'll look for song files by using matching filenames between the game level file and its corresponding song file.

Finding the song file

In the `game.lua` file, find the `scene:enterScene(event)` function that starts the game when it is completely ready, and attempt to load an audio stream from the same path:

```
function scene:enterScene( event )
self.Music = audio.loadStream(event.params.File:gsub('[^.]+$', 'mp3'))
  self.Velocity = 50
end
```

If the file cannot be loaded (say, because it does not exist), the attempt will return nil, and the next step can be skipped:

```
  self.Music = audio.loadStream(event.params.File:gsub('[^.]+$',
'mp3'))
  if self.Music then
  end
  self.Velocity = 50
```

Playing the song file

As long as the song file was loaded, we can go ahead and play it, saving the returned channel so that we can close it later if the song needs to be stopped prematurely (such as by the player failing the level):

```
  if self.Music then
    self.MusicPlaying = audio.play(self.Music)
  end
  self.Velocity = 50
```

Closing the song file

When the scene is exited, we need to stop the music in case it's still playing:

```
function scene:exitScene( event )
  if self.MusicPlaying then
    audio.stop(self.MusicPlaying)
    self.MusicPlaying = nil
  end
end
```

We'll finish cleaning up the audio stream object so it doesn't hold on to any memory:

```
    self.MusicPlaying = nil
  end
  if self.Music then
    audio.dispose(self.Music)
    self.Music = nil
  end
end
```

What did we do?

We loaded a song file from direct association to its matching game file, by changing the desired file extension. We loaded the file as a stream so that it would consume only a little memory at a time, and cleaned it up when the scene completed.

Enabling Game Center on the Provisioning Portal

To run or test Game Center code in your app, you have to create a custom App ID on Apple's Provisioning Portal and use it to build your app.

Getting on with it

First, you will need to log into your Apple Developer account on the iOS Dev Center at `https://developer.apple.com/devcenter/ios/index.action`.

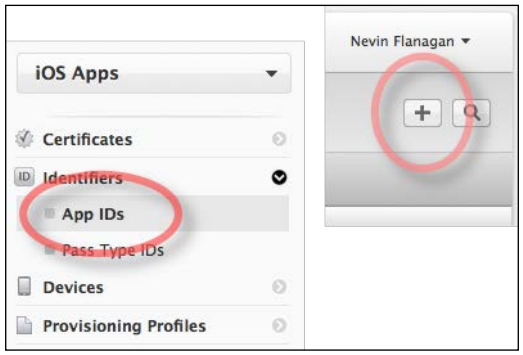
Create the App ID

Once you're logged in, the iOS Dev Center main page should appear.

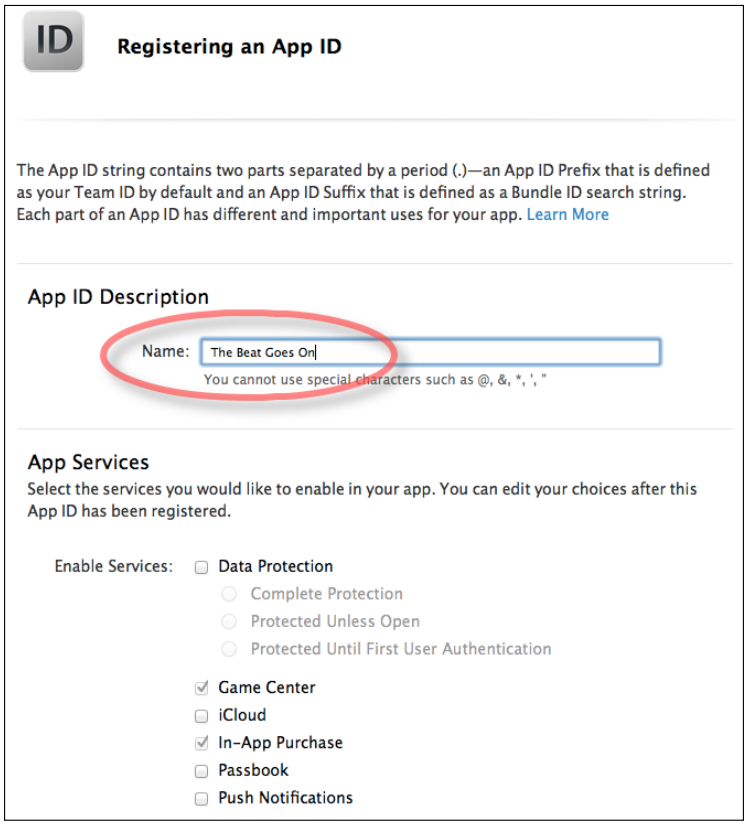
1. From the main page of the iOS Dev Center, open the Provisioning Portal:

2. Once on the Provisioning Portal, select the **Identifiers** link from the **iOS Apps** section and click on the **Identifiers** header again in the resulting screen to expand it:

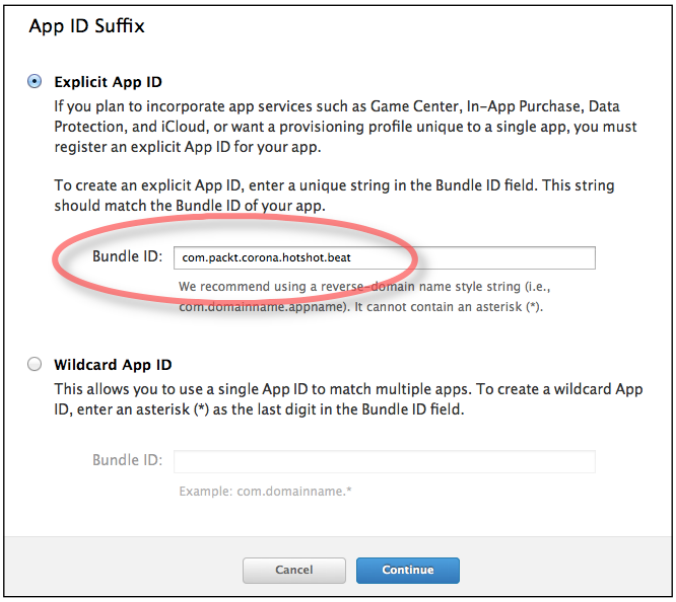
3. Once the header is expanded, click on the **App IDs** link and click on the button in the upper-right corner with the plus **+** icon to register a new App ID:



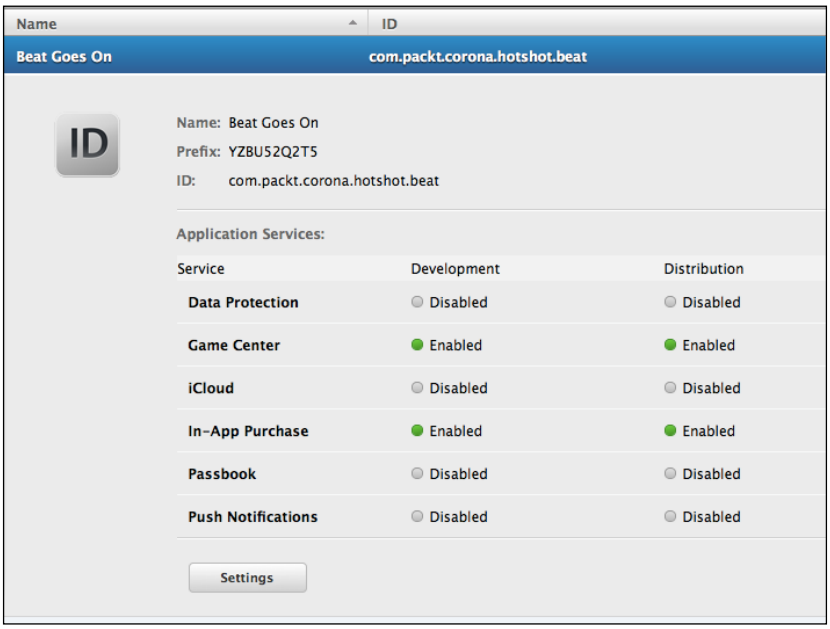
4. In the screen that pops up, first enter a descriptive name for your App ID, usually the name of your app:



5. Next, scroll down and enter a bundle ID, a reverse-domain-name format identifier specific to your organization, school, or studio:



6. After you click on **Submit**, you can see that your App ID is enabled for Game Center:



What did we do?

We created a tag that will allow Game Center to correctly associate our requests with the right set of leaderboards and achievements, once we create that set.

Enabling Game Center on iTunes Connect

Building your app using a Game Center-enabled App ID is the first step, but won't yield any useful results if there are no leaderboards or achievements to post to or unlock.

Getting ready

You should have already created an iTunes Connect account using the same Apple ID as your Apple Developer account. If you're still logged into the iOS Dev Center, there may be a direct link to iTunes Connect conveniently in the upper-right corner near the link for the Provisioning Portal; otherwise, you can always go there directly by directing your browser to `http://itunesconnect.apple.com`. You will also need to create application info for your app so that you can configure leaderboards or achievements using the bundle ID you created in the previous step.

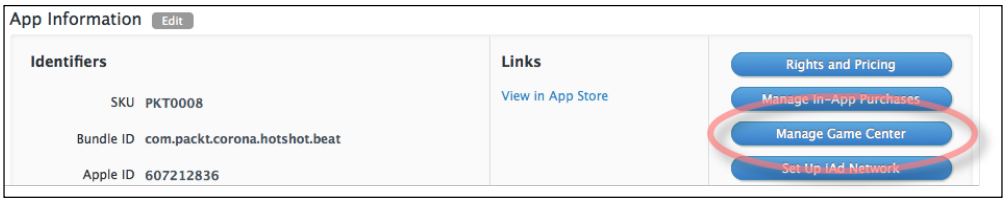
This means that you have to create your app info on iTunes Connect as soon as you want to test Game Center, rather than right before launch. However, regardless of what release date you have in mind, your app won't become available until you formally upload a binary.

Getting on with it

After logging into iTunes Connect, select **Manage Your Applications** and the info that you created for your new app.

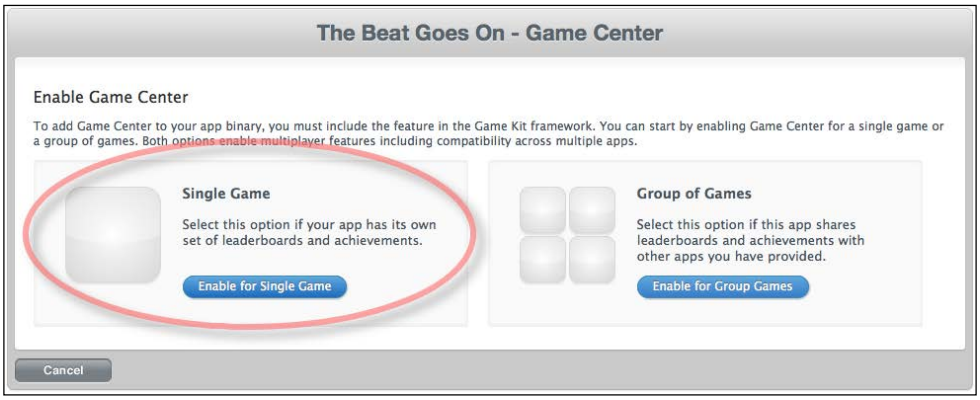
Enabling Game Center

Once you're looking at your app info, hit the **Manage Game Center** button on the right-hand side of the screen:



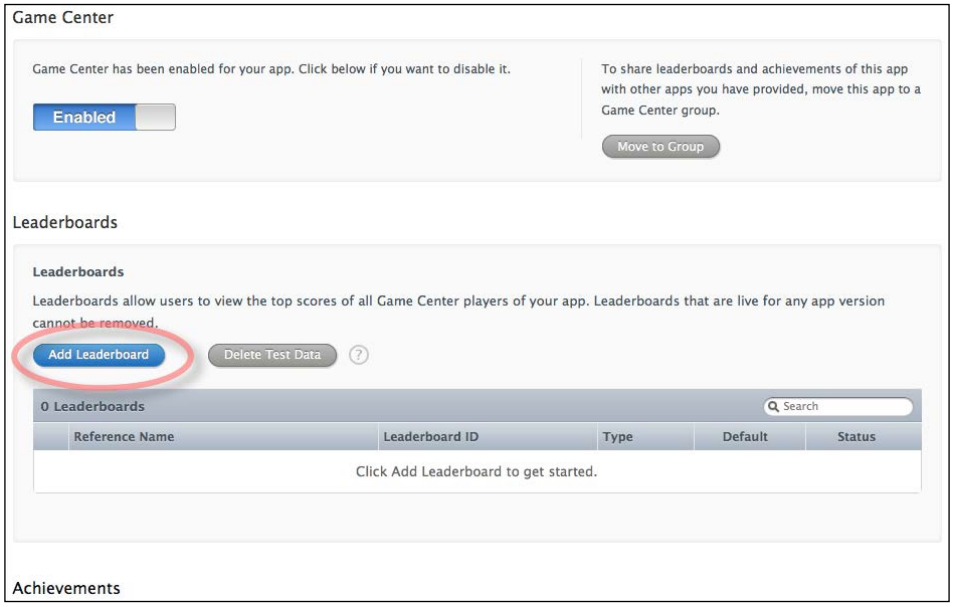
This presents you with a new choice: **Single Game**, or **Group of Games**. Choose **Single Game** in this case.

Select **Group of Games** if you have games that want to share leaderboards and achievements. The most common case for this will be if you release a free version as well as a paid version of your game.

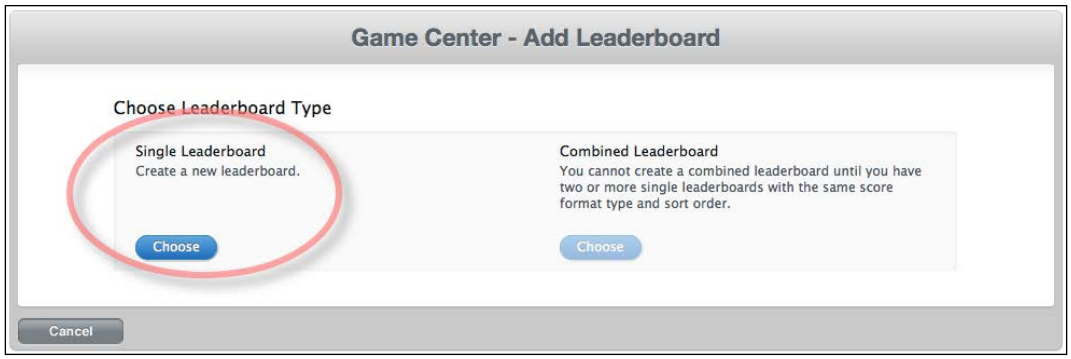


Adding a leaderboard

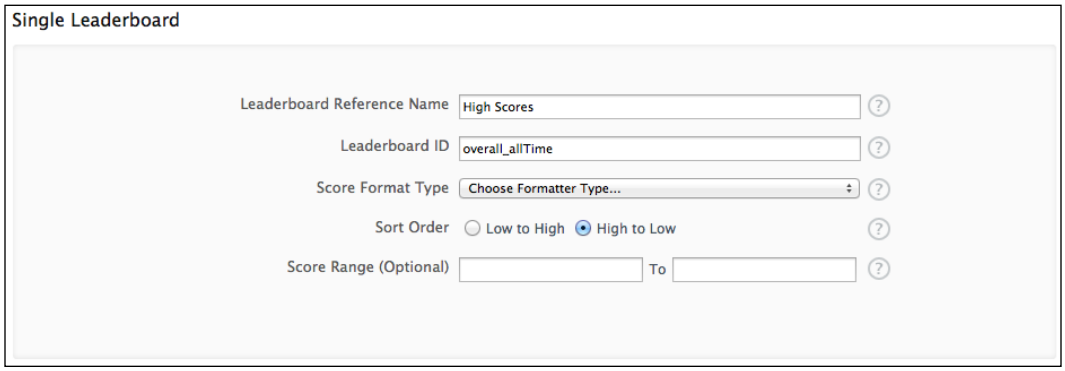
Once you select how Game Center is enabled, you should see a screen that lets you add new features. Select **Add Leaderboard**:



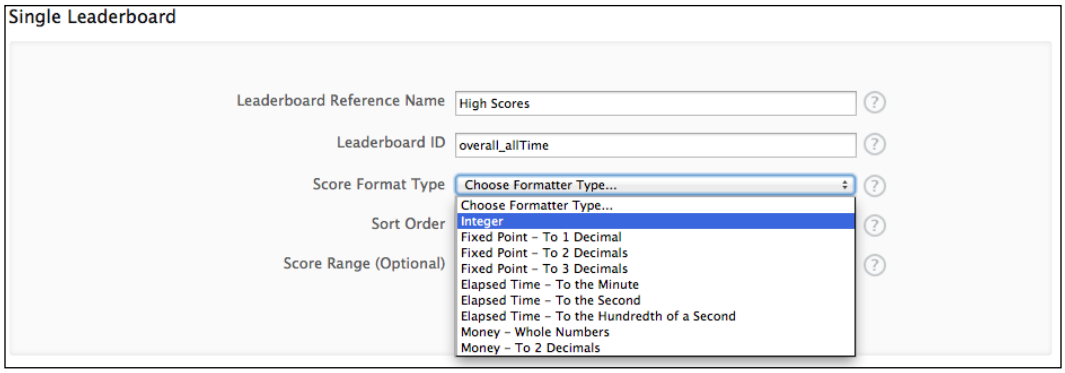
Next, you'll have to specify whether the leaderboard is a single set of values, or whether it's a combination assembled from several other leaderboards of the same kind (like single-player and multiplayer high scores). Choose **Single Leaderboard**:



You'll need to specify a type of value that will be ranked, and an ID string that your code will use to refer to this leaderboard. Enter `High Scores` and `overall_allTime`:



You can specify how the values should be formatted (a Monopoly style game, for example, might present scores in the form of money amounts), and whether scores should be ranked in descending or ascending order (the card game Hearts, for example, treats low scores as better). Pick **Integer**, and **High to Low**:



Finally, you have to select at least one localization. Localizations allow you to specify things like what sort of units the stored values are in (points, cards, seconds, dollars, and so on) as well as what the icon for the leaderboard should be (in case it includes text, or symbols specific to certain cultures or regions). Click on the **Add Language** button:

Choose **English** for the language, enter `All-Time High Scores` for the name, and pick **Integer (100,000,122)** for the score format (unless you live in Europe or another region where large numbers are grouped with periods and decimals are separated by commas, such as €132.436,84). Enter `point` and `points` as the singular and plural formats for the score suffix.

Finally, click on **Choose File** and browse for the `music.png` file included in the `original artwork` folder of the project pack. Once the icon appears, click on **Save** to close the Language dialog and click on **Save** again. Your leaderboard is ready for use!

Leaderboard icons need to be either 512 x 512 pixels, or 1024 x 1024.

What did we do?

We created a leaderboard associated with the ID that we generated in the last step. Once our app is built using that ID, any communications it makes with Game Center will automatically be connected with the new set of leaderboards (as well as achievements, if any).

Initializing a game network connection

Now that everything is enabled, the app itself needs to request a connection to the specified set of social features. It will use Corona's `gameNetwork` library to sign into the remote service, so that later calls will make requests to the right host.

Getting ready

Return to your project directory and open the `main.lua` file.

Getting on with it

Near the top of `main.lua`, select a module based on the operating system:

```
do
  local valid, loader = pcall(require, "social."..system.getInfo("plat
formName"):gsub("%s", "_"))
  if valid then
    social = loader
  else
    social = {loaded = false}
  end
end

local storyboard = require "storyboard"
```

Because `system.getInfo` can return a platform name containing spaces, the `gsub` call is there to avoid conflicts with tokens or other issues.

Registering application load

Inside the `social` folder in your project directory, create a new text file, `iPhone_OS.lua`, and open it for editing.

First, create a local table that can be used to share states with outside code:

```
local gameNetwork = require "gameNetwork"
local status = {}

return status
```

Register a system event listener so that we can connect after the rest of our app has finished loading:

```
local status = {}

local function loader(event)
end
Runtime:addEventListener('system', loader)

return status
```

Tracking connection progress

When Game Center finishes loading, Corona will call the function, if any, that you provided when you requested the initialization, and provide it with the results of the initialization attempt, using the value `init` for the `event.type` field when initialization is finished:

```
local status = {}

local function register( event )
  if event.type == 'init' then
  end
end

local function loader(event)
```

We can use this to record whether we connected to the service successfully, using the local table we recorded earlier:

```
local function register( event )
  if event.type == 'init' then
    status.loaded = event.data
  end
end
```

Requesting a Game Center connection

Finally, we'll combine this tracker function with the response we prepared to the application loading so that we actually request a connection to the game network:

```
local function loader(event)
  if event.type == 'applicationStart' then
    gameNetwork.init("gamecenter", register)
    Runtime:removeEventListener('system', loader)
  end
end
Runtime:addEventListener('system', loader)
```

What did we do?

We made a connection to the Game Center service once all the application resources were loaded and are now ready to receive it. We also left ourselves a way to determine whether or not this connection was completed.

What else do I need to know?

App functionality that depends on Game Center can't be tested in the Corona Simulator, because Game Center depends on your app being built with the correct bundle ID to connect it to the desired features (leaderboards and achievements). You need to build it for your device or for the XCode Simulator in order to test Game Center features.

Updating and reading a leaderboard

Now that our leaderboard is ready, we need to actually log our scores to it.

Getting ready

Save any other open files and open the file `game.lua`.

Getting on with it

Locate the `scene:Game(event)` function which responds to a game being over.

Checking network availability

To be able to submit high scores, we have to be connected to the network; otherwise, we should just leave the game:

```
scene:addEventListener('Game', scene)
function scene:Game( event )
  if social.loaded then
  else
    storyboard.gotoScene('splash')
  end
end
```

Submitting a final score

When a game ends, collect the scene's score variable and submit it to the game network:

```
  if social.loaded then
    gameNetwork.request('setHighScore',
      {
        localPlayerScore = {category = 'overall_allTime', value =
self.ScoreCurrent};
      }
    )
  else
    storyboard.gotoScene('splash')
```

Then add a listener so that the game can carry on once the scores are submitted and the leaderboard has responded that it has received them:

```
local function displayScores( event )
end

scene:addEventListener('Game', scene)
function scene:Game( event )
  if social.loaded then
    gameNetwork.request('setHighScore',
      {
        localPlayerScore = {category = 'overall_allTime', value =
self.ScoreCurrent};
        listener = displayScores;
      }
    )
  else
```

Displaying leaderboard values

In the listener for the score submission, we'll fall back on a built-in leaderboard display to show the results of the submitted score and how it ranks:

```
local function displayScores( event )
  gameNetwork.show('leaderboards',
    {
      leaderboard = {
        category = 'overall_allTime',
        timeScope = 'AllTime';
      };
    }
  )
end
```

This request will also get a listener, a simple closure that will send the game back to the splash screen once the user dismisses the leaderboard display:

```
  gameNetwork.show('leaderboards',
    {
      leaderboard = {
        category = 'overall_allTime',
        timeScope = 'AllTime';
      };
      listener = function( event )
        storyboard.gotoScene('splash')
      end;
    }
  )
```

What did we do?

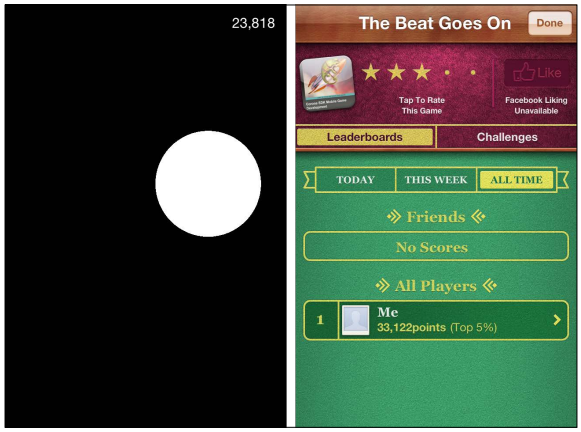
We used the results of our Game Center initialization call from the last step to determine whether scores could be sent to a leaderboard, then forwarded them if the leaderboard was available. Finally, we used Game Center's own built-in facilities to show the current status of the leaderboard before returning to the splash screen.

What else do I need to know?

In addition to the functions we've used here that ask the network itself to display its info, the `gameNetwork` package also includes requests that allow you to scan leaderboards yourself, collect data about other players, format your own lists of achievements, and so on. These allow you to take over some of the coding work in exchange for much more control in making the displayed leaderboards and achievements fit the appearance of your game.

Game over – wrapping it up

You've gotten a start on two major tasks in this project. A full discussion of multi-touch, state handling, and gesture recognition could very easily encompass a book of its own. Game Center and other game networks are not actually very complicated from the programming side; getting them working correctly is mostly a matter of configuring them correctly on the server support side.



Can you take the HEAT? The Hotshot Challenge

Now that you've come this far, try adding some achievements to the game. Some possibilities include making it all the way through your first song, getting a score above 100,000 points, or getting through 30 seconds of the song without missing a target. This will require both setting up the achievements on iTunes Connect much like you did with the leaderboard, and using the `unlockAchievement` string key with `gameNetwork.request`.

Project 9

Into the Woods – Computer Navigation of Environments

As players, most of us enjoy a game more when the ways in which computer characters move around seem more sensible or natural to us, rather than when they are obviously mechanical or over-simplified. Creating characters that can sensibly choose goals and find a good way to get there goes a long way toward improving a player's enjoyment.

What do we build?

We're going to take a tile-based exploration and adventure game, built from the same engine that we used in *Project 7*, *Caves of Glory*, and add calculated path finding so that enemies start to hunt you down when you get near, even around corners and obstacles.

What does it do?

The project will enable computer-controlled enemies in the game to respond when the `player` character comes within their range, by selecting a way to reach the player and travel until they reach him or her. This path finding means that as long as the player stands still within range, monsters alerted to his/her presence will eventually reach him/her; he/she can't rest at any spot where enemies would simply get stuck in a corner trying to reach him/her.

You'll add this to the project by writing code that carries out what is called the **A*** algorithm (pronounced *ey-star*), an approach to finding the easiest available path between where a character is and where it's going. A* takes an account of whether certain areas between here and there are harder or take longer to cross, and it uses "off-the-cuff" estimates of the likely distance to focus on finding a good, usable way to reach the target fast. It's not guaranteed to find the best possible way to reach its destination, but as long as you provide it with reasonable guidelines, it will never prefer a lengthy awkward route when a substantially better one is nearby.

Why is it great?

Firstly, familiarity with A* can only help you if you want to develop smart game behaviors—it's powerful, commonly known, and well-understood. Its balanced tradeoff of good results against fast performance has made it the go-to choice of path-finding scheme for many games applications.

The other step that this project takes is to really illustrate how powerful projects start to become when you can re-use techniques from other examples. The `world` and `map` engines are built on top of the code from *Project 7*, *Caves of Glory*, with only a few modifications. The code that controls both computer-controlled and player-controlled characters is based on the AI controllers from *Project 5*, *Atmosfall*. The touch controls to move your character are taken almost directly from *Project 2*, *SuperCargo*. Being able to assemble components that can easily be modified and rearranged, much like toy building sets, is where the real power of good software creation starts to appear.

How are we going to do it?

Unlike previous projects, most of this project will be focused on pieces of a single algorithm. We will perform the following operations to do so:

- Structuring the A* algorithm
- Sorting likely routes using a `heap` form
- Writing a custom iterator
- Selecting costs for neighboring tiles
- Writing a heuristic function
- Using the implementation to find a path
- Moving based on path selection
- Following the main character

What do I need to get started?

As with other projects, you will need to copy the contents of the `version 0` folder in the project pack into your own project folder called `IntoTheWoods`.

You may also want to try the demo at `http://www.antimodal.com/astar/`. Set the **steps per frame** value fairly low, to `1` or `2`, and hit **Find Path** to see the algorithm in action. Watch how it occasionally gets backtracked but finds a short path after searching less than a quarter of the area.

Structuring the A* algorithm

We'll lay out the basic structure of the A* search, looking for the point of the current search border likeliest to be closest to the destination, checking to see if any of them offer a faster route to known spaces, and whether or not we have reached our target.

Getting ready

Create a new file, `path.lua`, and open it for editing.

Getting on with it

We'll start by laying out the generic skeleton of an A* implementation.

Assembling the requirements

In addition to a start position, a goal position, a progress function, and a heuristic function, A* needs to create two things internally to do its job—a list of spaces currently on the border of its search, and the known costs to reach them, and a list of the spaces that the search has determined which are in the best path to any given known space using the following code:

```
return function(start, goal, neighbors, h)
  local costs = {[start] = 0}
  local parents = {}
end
```

Selecting the cheapest choice

The `costs` table holds all spaces that have been considered at all as part of the path, but still have any unknowns about the paths going through them. We need to select whichever space in the list has the lowest projected total cost using the following code:

```
return function(start, goal, neighbors, h)
  local costs = {[start] = 0}
  local parents = {}
```

```
  for prospect in lowestEstimate(costs, function(source) return
h(source, goal) end) do
  end
end
```

We'll cover the details of the `lowestEstimate` iterator in a few steps.

Checking for completion

As shown in the following code, if the easiest space to reach under consideration *is* the goal space, well, we found it!

```
  for prospect in lowestEstimate(costs, function(source) return
h(source, goal) end) do
    if prospect == goal then
    end
  end
```

Then we just need to unravel the chain of parentage back to its beginning using the following code:

```
    if prospect == goal then
      while parents[prospect] and parents[prospect] ~= start do
        prospect = parents[prospect]
      end
    end
```

As shown in the following code, whichever node along the path was reached from the start position is the first step that needs to be taken to follow that path:

```
    if prospect == goal then
      while parents[prospect] and parents[prospect] ~= start do
        prospect = parents[prospect]
      end
      return prospect
    end
```

Advancing the search

If we haven't reached the goal, then we need to take the spaces we can reach from this one, and the costs to move to them from this point.

```
      return prospect
    end
    for successor, cost in neighbors(prospect) do
    end
  end
```

The `neighbors` iterator in the previous code snippet is used in a generic form. The algorithm doesn't care *how* neighbors are determined or the costs to reach them are calculated, as long as the iterator returns a neighbor node and a cost to travel to that node for each adjacent (or otherwise reachable) space, and returns `nil` when it has shared every possible neighbor with us.

We'll create our own specific `neighbors` function for use with our maps in the next task.

Registering new or improved possibilities

For each neighbor, we'll consider whether it's either a new discovery, or whether the path we took to get here represents a new shortcut over the currently known path:

```
for successor, cost in neighbors(prospect) do
  cost = cost + costs[prospect]
  if not costs[successor] or costs[successor] > cost then
  end
end
```

The next line is a bit quirky. Of all the steps in the whole process, it's probably the most artificial; it exists entirely because of how we'll maintain our search structure, a couple of tasks down the road.

```
if not costs[successor] or costs[successor] > cost then
  if costs[successor] then costs[successor] = nil end
end
```

The previous highlighted line exists just to make sure the following line works the way we expect, by guaranteeing that each assignment into the `costs` table is sorted correctly for the next pass through the loop.

```
if not costs[successor] or costs[successor] > cost then
  if costs[successor] then costs[successor] = nil end
  costs[successor] = cost
end
```

If we either added this new space for consideration, or determined that this is a better way to reach the target neighbor than any previously known, we need to remember that this prospect is the best known way to reach the space in question.

```
if not costs[successor] or costs[successor] > cost then
  if costs[successor] then costs[successor] = nil end
  costs[successor] = cost
  parents[successor] = prospect
end
```

This is in fact the entire form of the A* algorithm, with only three major details needed to make it fully functional.

What did we do?

We laid out the basic skeleton of the A* methodology—a sort of rubber band constantly being stretched out in the direction of the goal, held back by areas that are harder or impossible to pass.

What else do I need to know?

A* is an extension of a method called Djikstra's algorithm, after the influential computer scientist Edsger Djikstra who published it in 1959. Djikstra's algorithm takes a starting point and finds the shortest route from there to every other point you can get to. Because it covers the entire map, it can take a while to run, especially with large maps; A* expands it by considering not just the easiest next step, but the one that leads to the next space with the best combination of known cost to get there plus likely cost to get the rest of the way. This means that A* keeps moving in generally the right direction.

Writing a custom iterator

Most `for` loops in Lua are designed to process a list of things, most often some or all of the keys in a table. These lists are produced by functions that get called repeatedly to produce one item at a time from the list.

Most of the time, these loops are set up using the standard functions `pairs` or `ipairs`. However, Lua allows you to supply any custom setup that meets a particular format, which we'll use to operate on each of the squares adjacent to a given square in the map.

We've actually dabbled in this before, in *Project 8*, *The Beat Goes On*, when we used a binding to return the results of `ipairs` on a specific table, but now we're going to expand this to create our own iteration processes.

Getting ready

Save `path.lua` and open `map.lua`.

Getting on with it

We're going to use a variant of the technique we used in *Caves of Glory*, to detect a character in the spaces next to a treasure chest.

Starting the loop

Add a local function to the beginning of the `map` module:

```
  return function() return head, s, v end, function() return f, s,
bridge end
end
local function nextNeighbor(start, previous)
end

return function (filename)
```

The loop following manages the offset from the start position in the center, defaulting to start from the position to the right of the start (although since it advances once at its start, it internally starts from the position above and moves once clockwise):

```
local function nextNeighbor(start, previous)
  local dX, dY = 0, -1
  repeat
    local row = start.map[start.y + dY]
    local space = row and row[start.x + dX]
    dX, dY = -dY, dX
  until dY = -1
end
```

The loop skips over blank spaces, such as would be found at the edges of the map:

```
    dX, dY = -dY, dX
    local row = start.map[start.y + dY]
    local space = row and row[start.x + dX]
    if space then
      return space
    end
```

Advancing the loop

Each time a `for` loop calls its generator function, it passes back the first value returned by the last call. We can use this to get the last offset positions as shown in the following code:

```
  local dX, dY = 0, -1
  if previous then
    dX, dY = previous.x - start.x, previous.y - start.y
    dX, dY= -dY, dX; if dY==-1 then return end
  end
  repeat
    dX, dY = -dY, dX
```

Preparing a loop

Create a new `local` function after the last one using the following code:

```
    until dY = -1
end
local function neighbors(start)
end
return function (filename)
```

This `local` function will prepare a new `for` loop when it is called, by supplying the `nextNeighbor` generator function and the space to use as the center, passed as an argument by the A* loop:

```
local function neighbors(start)
  return nextNeighbor, start
end
```

Go down to the contents of the main module function in this file, and add a new function field to the `map` module once it is constructed:

```
        end
        space.Neighbors = neighbors
        function self:Find(item)
```

The previous function is not a method and doesn't use colon (`:`) syntax.

The previous code exposes the new iterator to the outside world.

What did we do?

We kept our code readable and maintainable by keeping it in a familiar, intuitive structure (the `for` loop), facilitating this by using our own function to walk from one adjacent square to the next.

What else do I need to know?

The way Lua `for ... in` loops work is that they take three components—a generator function that produces the next value from the previous value, a domain for the iteration which stays constant throughout the loop (the table being scanned, for instance, in cases such as `pairs` and `ipairs`), and an initial value that is fed to the first call to the generator. The generator function itself will be called with two arguments, the iteration domain, and the first result from the last pass through the loop.

Rather than entering all three of these values yourself, you typically use a function that generates them accordingly. For instance, the `pairs(t)` function actually returns three values—the built-in function `next`, the table `t` itself, and `nil`.

Selecting costs for neighboring tiles

The other piece of information that we need to know about each neighboring tile is how hard, compared to other tiles, it is to take the step into that tile from the one we are considering. We'll add this information to the terrain engine and attach it to the spaces to make it easy to find.

Getting ready

Open the `forest.lua` file in the `terrain` folder.

Getting on with it

This model will keep things simple, basing costs to enter a space solely on the type of space being entered. This means we can use a simple table.

Providing cost assignments

Add a new local table near the top of `forest.lua`:

```
local forest = {}
local entryCosts = {
}
function forest:Expand(kind)
```

Add values for the basic terrain types found in the map:

```
local entryCosts = {
  bush=
  math.huge,
  ['=grass']=1,
}
```

The use of `math.huge` means that the pathfinder will prefer any path over grass and rocks, no matter how long, over one that goes through trees. We can also test whether the length of a path is `math.huge` to avoid trying to follow impossible paths.

Selecting costs for terrain types

We'll also provide a function that selects costs from the following table as needed:

```
  };
};
Cost-function(start,neighbour)
end;
Polish=function(tile)
```

Polished terrain maps may have variants on the same terrain type, so we use only the core type; terrain types are stored as strings where any subcategory comes after the basic terrain type, separated by a colon:

```
function forest.Cost(start, neighbor)
  local name = neighbor.Ground
end
```

Finally, we return the appropriate value from the table, with a default that ensures unrecognized terrain types will not be crossed:

```
function forest.Cost(start, neighbor)
  local name = neighbor.Ground
  return entryCosts[name] or math.huge
end
```

Save `forest.lua`.

Forwarding the cost data

Return to the `nextNeighbor` function in `map.lua`. Remember from `path.lua` that the iterator needs to return two values, the next neighbor and the cost, like the following iterator code does:

```
    local space = row and row[start.x + dX]
    if space then
      return space, start.map.Terrain.Cost(start, space)
    end
  until dY = -1
```

What did we do?

We completed the neighbor analysis process by adding weights to different tile transitions, allowing the different terrain types to offer specifics about what type each square represents.

What else do I need to know?

This cost-assessment function is not terribly efficient. Since it is called many times for each pass through the map, if there are many monsters, all calculating a path regularly, this inefficiency will be multiplied enough to have a serious impact on game performance.

One way to make it more efficient is to precalculate each square's neighbors and costs, since they don't change. You can store a table in each map square that records the neighbors and their costs, such as by using neighbor square tables as keys and costs as the values. Scanning for neighbors then becomes a matter of simply iterating that table.

Sorting likely routes using a heap

The A* routine spends a lot of time selecting the space with the shortest known route; because it does this over and over, if we use an inefficient way to find this space in the list, it will rapidly add up and slow down the processing, interfering with the game frame rate. To keep performance high, we store our list in a form that computing scientists call a **heap**, which can be kept sorted in an efficient way as things are added to it or taken away.

Getting ready

Save and re-open `path.lua`. Note the point where the main for loop of the A* function uses the `lowestEstimate` function to iterate over available spaces. This is another customer iterator; it will set up a heap based on the `costs` table we define, and then repeatedly remove the value with the lowest supposed cost.

Getting on with it

A heap is a way of organizing an array so that each element is always sorted before the two regarded as its children (according to the sorting of your choice). While the array is not sorted from front to back, it eliminates half of the remaining array with every sort step it performs, making the sort process very fast. You can see the end of the task for more detailed information on how heaps and the heap-sort process works.

Building the heap form

Frame in the `lowestEstimate` function at the top of `path.lua`. This function receives the set of costs, as well as the heuristic function that can be applied to each space.

```
local function lowestEstimate(costs, h)
end
return function(start, goal, neighbors, h)
```

The following function will build an internal `heap` form that sorts the different elements in the costs table according to their total estimated travel costs, which will be stored in another table:

```
local function lowestEstimate(costs, h)
  local estimates = {}
  local heap, length = {}, 0
  for n, c in pairs(costs) do
  end
end
```

For the initially supplied costs, we'll expand them based on a list of heuristic values (which we'll assemble in a moment), adding items into the heap without initial regard for sorting:

```
local function lowestEstimate(costs, h)
  local estimates = {}
  local heap, length = {}, 0
  for n, c in pairs(costs) do
    estimates[n] = c + heuristics[n]
    length = length + 1
    heap[length] = n
  end
end
```

The A* algorithm assumes that a space's heuristic value never changes during the course of the algorithm.

Once all items are added, the `heap` form is sorted from the bottom-up. This will tend to get potentially redundant work out of the way early:

```
    heap[length] = n
  end
  for subtree=(length - length % 2) / 2, 1, -1 do
    pushdown(heap, subtree, length)
  end
end
```

Adding to the heap form

For the `heap` form to be useful for sorting a selection, we have to be able to add new items to it, but our iterator doesn't provide access to the `heap` form directly, because it wants to keep it sorted. However, we can use a metatable attached to the original `cost` set, to record whenever a new value is added to the set and fix the `heap` form as shown in the following code:

```
    pushdown(heap, subtree, length)
  end
  local function relist(costs, node, cost)
```

```
    end
    setmetatable(costs, {__newindex = relist})
  end
```

The first thing we need to do is actually record the new cost. The `rawset` function works like the statement `costs[node] = cost`, but without triggering the metatable, which would cause an infinite loop:

```
local function relist(costs, node, cost)
  rawset(costs, node, cost)
end
```

Then we need to record the estimated total cost for the new value, the way we did when we built the `heap` form:

```
local function relist(costs, node, cost)
  rawset(costs, node, cost)
  estimates[node] = cost + heuristics[node]
end
```

If the element has been in `heap` before, we find its old place before resorting it. Otherwise, we add it to `end`:

```
local function relist(costs, node, cost)
  rawset(costs, node, cost)
  estimates[node] = cost + heuristics[node]
  local i = table.indexOf(heap, node)
  if not i then
    i = #heap + 1
    heap[i] = node
  end
end
```

Once it's in the `heap` form, we push it up toward the top, swapping it with its parent, until it's in the correct sorted position:

```
    heap[i] = node
  end
  local parent = (i - i % 2) / 2
  while parent >= 1 and estimates[heap[parent]] > estimates[node] do
    heap[parent], heap[i] = node, heap[parent]
    i = parent
    parent = (i - i % 2) / 2
  end
end
```

Caching heuristic values

Because heuristics don't change over the course of the algorithm, we want to avoid recalculating them. But the purpose of this method is efficiency, so we also don't want to calculate a value for any node that never gets considered. We can solve this with another metatable-based pattern, very common in Lua, called a **self-populating table**:

```
local estimates = {}
local heuristics = setmetatable({},
  {
    __index = function(t, node)
    end
  }
)
local heap, length = {}, 0
```

The `index` function for this table is called whenever a value is indexed that doesn't exist in the table yet. First, it calculates the heuristic for that node.

```
__index = function(t, node)
  local val = h(node)
end
```

It saves the value in the table, so that the `__index` function won't be called again when the value is requested later:

```
__index = function(t, node)
  local val = h(node)
  t[node] = val
end
```

And it returns the value, so that it can be used when it was requested:

```
__index = function(t, node)
  local val = h(node)
  t[node] = val
  return val
end
```

Fixing heap

The biggest complexity in the `heap` function is the process of sorting a value to the bottom of its sub-tree. We've used the `pushdown` function already, but it needs to be implemented as shown in the following code snippet:

```
local estimates = {}
local function pushdown(heap, start, l)
end
local heuristics = setmetatable({},
```

The function keeps checking the value against its children until neither one is out of order with it:

```
local function pushdown(heap, start, l)
  repeat
  until not branch
end
```

If one is out of order with it, it has to swap with whichever child can become the parent of the other and stay sorted:

```
  repeat
    local leftChild = start * 2
    local smaller, branch = math.huge, nil
    if leftChild <= l then
      smaller, branch = estimates[heap[leftChild]], leftChild
      local rightChild = leftChild + 1
      local value = estimates[heap[rightChild]]
      if rightChild <= l and value < smaller then
        smaller, branch = value, rightChild
      end
    end
  until not branch
```

Once it has determined which child can be selected to move up, it checks whether they are, in fact, out of order:

```
      if rightChild <= l and value < smaller then
        smaller, branch = value, rightChild
      end
      value = estimates[heap[start]]
      if smaller < value then
      end
    end
```

If so, it swaps them to restore order and checks the new sub-tree:

```
      if smaller < value then
        heap[start], heap[branch] = heap[branch], heap[start]
        start = branch
      end
    end
```

Otherwise, if the element is in order with both children, `heap` is sorted again and we can stop:

```
        if smaller < value then
          heap[start], heap[branch] = heap[branch], heap[start]
          start = branch
        else
          break
        end
```

Pulling a value

To iterate over the list, we have to be able to get the smallest `cost` value and take it out of the list. At the bottom of the `lowestEstimate` function, start constructing the generator function that will do this. This function will also need the `heap` object as its iterator state.

```
  setmetatable(costs, {__newindex = relist})
  return function(t, ...)
  end, heap
end
```

Getting the lowest estimate is easy; it's at the beginning of the `heap` object.

```
  return function(t, ...)
    local root = t[1]
    return root
  end, heap
```

As long as `heap` was not empty, we need to fix it. The easiest way to do this is to grab the last element in the whole `heap` object, jump it up into the gap, and then sort it down using our existing fixer:

```
  return function(t, ...)
    local root = t[1]
    if root then
      local length = #t
      t[1] = t[length]
      t[length] = nil
      if length > 2 then
        pushdown(t, 1, length - 1)
      end
    end
    return root
  end, heap
```

There! Our `heap` sort is ready to organize our selected spaces. The A* module is almost ready.

What did we do?

We kept our algorithm efficient by organizing our selection of spaces under consideration into a sorted tree arrangement, or heap; this makes it easy to add new options or remove old ones while keeping them sorted in a reasonably short time.

What else do I need to know?

Heaps are a non-trivial piece of computing, and we don't completely cover how they work. The basics are that they store a tree inside an array, to keep it compact and simplify the process of allocating searching for parents and children. If the root of the tree is element 1 of the array, then each element *N*'s children are element 2**N* and element 2**N* + 1.

When a new element is added to `heap`, it is added at the very end, pushed up the tree until it reaches a parent that does not sort earlier than it, and then, if needed, pushed back down another branch of the tree until it is sorted properly. When an element (usually the first element) is removed, the last element of `heap` is swapped into its place, and then pushed back down the tree until it settles into a properly sorted location. All of this is designed to keep the number of swapped or moved elements to an absolute minimum.

You can find a step-by-step example of a heap sort in action at `http://ds-algo.blogspot.com/2007/06/heap-sort-explained.html`. A technical explanation of heap sorting and the principles that make it work can be read at `http://en.wikipedia.org/wiki/Heapsort`.

Writing a heuristic function

As mentioned earlier, A* is an evolution of a slightly simpler algorithm called Djikstra's algorithm, which is guaranteed to find the shortest available path, but takes longer to run. The principle difference is that A* uses a heuristic function that gives its best guess about which squares are closer than which others to the destination. In this way, it tries to keep its focus moving in the general direction of the goal. We'll choose a reasonable heuristic for the type of map we're using, and implement it so that it will be available to in-game creatures of a certain type.

Getting ready

Create a new file, `walker.lua`, and open it for editing.

Getting on with it

Since our map is a square grid, for our heuristic, we'll use what is called the "Manhattan" distance, the square distance that has to be counted along both axes to get to the goal.

Constructing the function

Frame the function in at the top of the new file:

```
local function manhattan(space, goal)
end
```

The function will take both the `x` and `y` distances and add them together:

```
local function manhattan(space, goal)
  return math.abs(goal.x - space.x) + math.abs(goal.y - space.y)
end
```

What did we do?

We added a "Manhattan" estimation of distances between any arbitrary square on the map and a selected goal square in the same map, allowing our path-finding process to reach a reasonable result more quickly.

What else do I need to know?

A*'s performance is highly dependent on your choice of heuristic. If your heuristic has a very high weight compared to your `cost` values, the algorithm will tend to find a route very quickly, but at the possible expense of skipping over better routes nearby. If your heuristic is weighted very low, the algorithm will search more possible routes, typically finding a better route from the available choices but taking longer to consider more options. If your heuristic is zero, the algorithm becomes Djikstra's algorithm, searching out every possible route but always returning the best one possible.

On grid maps, the Manhattan distance (so called because the borough of Manhattan in New York City is laid out very neatly in a grid pattern) is a fairly reliable heuristic; on free-form maps, the Pythagorean distance, or radial distance, is a good start. On large, complex maps, a more sensitive function may be needed to keep performance up; for instance, regions of the map might be assigned weights according to the prevailing terrain. A swampy region might be generally assigned a high heuristic cost, even if low-cost paths through it exist, to indicate that only the most useful shortcuts should be pursued.

Connecting all the elements

All of the pieces for our path-finding implementation are ready now. Our next step is to bind them all together into a module that can determine the shortest path when traveling "on foot" through our virtual environment.

Getting ready

Open `walker.lua`, if needed.

Getting on with it

The walker module will specify which neighbor and heuristic functions need to be used with the algorithm.

Assembling dependencies

The module will therefore need to load the path module so that it can call our A* function.

```
local path = require "path"
local function manhattan(space, goal)
```

Binding the components

The module `function` will take a start space and an end space, and use the `path` module to pick a step in the right direction.

```
  return math.abs(goal.x - space.x) + math.abs(goal.y - space.y)
end
return function(start, goal)
  return path(start, goal, start.map.Neighbors, manhattan)
end
```

What did we do?

We've given the A* algorithm a particular case to work on, by choosing functions that provide it with neighbors and a heuristic for a specific kind of map. If we wanted to create a flying or swimming creature, we would need to replace the `neighbors` function to modify the costs for obstacle spaces or areas of water.

Using the implementation to find a path

Now that the path-finding implementation is complete, it's time to call it for any enemy who isn't currently doing something.

Getting ready

Open the file `monster.lua` and find the nested function called `idle`.

Getting on with it

The `idle` function does nothing right now; it just waits forever.

Monitoring while idle

Functions in these AI controllers tend to be very high-level and abstract. We'll use a provided iterator to look at each of the surrounding spaces within a certain distance:

```
function idle()
  while wait() do
    for space in surroundings(4) do
    end
  end
end
```

For each nearby space, we'll check its contents:

```
  while wait() do
    for space in surroundings(4) do
      for _, content in ipairs(space.Features) do
      end
    end
  end
```

If anything in the space is a character with a different allegiance, or faction, it's time to chase after it and attack:

```
    for space in surroundings(4) do
      for _, content in ipairs(space.Features) do
        if content.Character and content.Alleigance ~= self.
Alleigance then
          return hunt(content)
        end
      end
    end
```

Responding with pursuit

Add another function next to `idle`:

```
local idle, hunt
function hunt(quarry)
end
function idle()
```

These enemies are relentless and never stop chasing you due to the following code:

```
function hunt(quarry)
  while true do
  end
end
```

A hunting enemy checks whether its quarry is adjacent to it:

```
  while true do
    local dX, dY = location()
    dX, dY = dX - quarry.Location.x, dY - quarry.Location.y
    if math.abs(dX + dY) == 1 and dX * dY == 0 then
    end
  end
```

If so, the monster attacks (this feature isn't actually implemented yet):

```
    dX, dY = dX - quarry.Location.x, dY - quarry.Location.y
    if math.abs(dX + dY) == 1 and dX * dY == 0 then
      attack(quarry)
    end
```

Otherwise, the monster identifies the next step in getting to that point, and starts moving there. If the space is unavailable, step returns false, and the monster waits:

```
    if math.abs(dX + dY) == 1 and dX * dY == 0 then
      attack(quarry)
    else
      local proceed = path(location(), quarry.Location)
      if proceed then
        if not step(proceed) then wait() end
      end
    end
```

What did we do?

We tracked which computer characters are currently in the middle of an action, and when they're not already occupied with an attack or a movement, we've used this path-finding tool to select a course for them to follow to guarantee that they remain a threat to any player who stands and waits.

Moving based on path selection

Now that this path is being found as needed, the last crucial step is to actually carry it out.

Getting ready

Open the file `character/init.lua`, which includes specific actions and features used by both computer- and human-controlled players.

Getting on with it

The `step` function is in fact very similar to the network functions from TranslationBuddy, and that's what we're going to model it on:

```
function presence.step(target)
  local origin = getfenv(2).location()
  local self = coroutine.running()
  local function resume(event)
   event.target:removeEventListener('Moved', resume)
   coroutine.resume(self, 'finished')
  end
  local motion = origin.map:Move(character, getfenv(2).world, target.x
- origin.x, target.y - origin.y)
  if motion then
   motion:addEventListener('Moved', resume)
   repeat local outcome = coroutine.yield()
   until outcome == 'finished'
   return true
  end
 end
```

What did we do?

We expanded the computer character control code to carry out an actual `Move` command over time, and return control to the character controller when the move is completed.

Game over – wrapping it up

Path-finding is a sophisticated topic; while a basic A* implementation like this one is a robust solution to simpler maps under constraints such as tiled movement, more free-form movement environments will demand more elaborate solutions. However, the solution presented here is both fairly efficient and quite adaptable in the way you can redefine its neighbor and heuristic functions; for instance, you could allow different characters to navigate the same map by different routes if one of them can fly over cliffs and trees.

Can you take the HEAT? The Hotshot Challenge

Use one of the other monster sprites, such as the swamp monster, and create a new monster that moves through streams and other water, but moves more slowly over land. Use a `neighbor` function that prefers water over land as a result. You'll need to use additional art files from the original art folder to add water to the environment.

Project 10

Underfoot – Selectively Leveraging the Physics System

So far, we've used Corona's support for the Box2D physics system primarily as a collision detector. It's been used in three projects and we've immediately turned off the gravity in every case. In at least one case, most bodies we create are sensors. This project will explore how to use more of the actual physics-based features to build a platform game with gravity and solid floors and walls.

What do we build?

For this project, we will start with a partly completed platform game; it contains all sprites, and a tile map module similar to that used in *Project 7*, *Caves of Glory – Mastering Maps and Zones* and *Project 9*, *Into the Woods – Computer Navigation of Environments*. We will add physics to this world to create the world interactions that support game play; walls and floors that stop the player, collision detection with enemies that doesn't allow them to actually push the player or cause things to bounce off each other, and climbable ladders. We'll use collision filtering, in combination with sensor body elements, to keep only the physics interactions that are actually useful to us.

What does it do?

You'll start out with working modules to load a map from a file, load characters with sprites and accept control signals for them, supply control events for your character from user input, track score, and recognize level completion. The game implements a simple platformer, where you jump from block to block, score points by jumping on top of enemies to squish them, and navigate ladders and pools of water. The player will move the character to the left-hand side or right-hand side by holding the screen on the desired side, climb by tilting the device backward or forward, and jump whenever they release the screen (including a tap for standing jumps).

Why is it great?

This project will address one aspect of the physics module that we've let slide up to this point: **multi-element bodies** (fixtures in Box2D's internal parlance). We've used them a little in the shooter projects, solely as a way to create concave physics shapes without freaking out Box2D. However, in this case, we'll really go down the fixtures rabbit hole, using different collision filters for different fixtures as well as creating bodies that contain both solid fixtures and sensor fixtures.

How are we going to do it?

The tasks for this project will focus on controlling physics interactions.

- Building physics for the map
- Making characters interact with the world
- Driving character behavior
- Responding to collisions with other characters
- Bouncing off enemies as appropriate
- Controlling gravity to enable climbing
- Selecting collisions by manipulating contacts
- Adding polish with custom fonts

What do I need to get started?

You will need to copy the contents of the `version 0` folder in the project pack directory into a new project folder. These directions assume that you call this directory `Underfoot`.

Building physics for the map

In order to use physics to drive our game interactions, we'll need some physics content for our game world to give characters some floors (and walls) to interact with.

Getting ready

Open the file `categories.lua` and locate the table being returned (currently empty).

Getting on with it

We'll create a long physics block for every unbroken row of block tiles.

Defining the floor category

In `categories.lua`, add a new entry to the table being returned:

```
return {
  ground = {
    groupIndex = ID.general;
    categoryBits = ID.wall,
    maskBits = ID.sprite;
  },
}
```

Save this file, open `world.lua`, and load this module at the top of the file.

```
local categories = require "categories"

local function matchPointPlacement(target, referenceX, referenceY,
frame, targetX, targetY)
Since the module is working with collisions, it will also need
physics.
Local physics = require "physics"
local categories = require "categories"
```

Clearing the map

Open the file `world.lua` and locate the `self:Load(map)` function. After looping over the map, make sure that in case the world object is being reused, any previous platform objects are cleared:

```
    end
    if self.Blocks then
      for _, block in ipairs(self.Blocks) do
        display.remove(block)
```

```
    end
  end
  self.Map = map
```

Then, make sure that the world starts with its list of solids being newly empty:

```
      display.remove(block)
    end
  end
  self.Blocks = {}
  self.Map = map
```

Scanning the rows and identifying block runs

Look over each row of the map at a time:

```
  self.Blocks = {}
  for h, row in ipairs(map) do
  end
  self.Map = map
```

Then, for each row, gather the position, ground style, and width of each run of blocks using a custom iterator (which we'll write next):

```
  for h, row in ipairs(map) do
    for start, kind, width in scanRow(row) do
    end
  end
```

Before using these dimensions to create the blocks, we'll back out for a moment to create the iterator that will treat the map row as a sequence of blocks. Move back up the file before the module function and frame in the form of an iterator that works with numeric indices:

```
    until dY == -1
end

local function advance(row, start)
end
local function scanRow(row)
  return advance, row, 0
end

return function (terrain, columns, rows)
```

When the generator comes back to a row at the start or middle of a run of ground blocks, it scans to the end of that run before proceeding:

```
local function advance(row, start)
  while row[start] and row[start].isGround do
    start = start + 1
  end
end
```

It then advances until it finds a block that contains solid ground:

```
    start = start + 1
  end
  repeat
    start = start + 1
  until row[start].isGround
end
```

If it reaches the end of the row first, scanning is complete and it ends the loop by returning nil:

```
  repeat
    start = start + 1
    if not row[start] then
      return
    end
  until row[start].isGround
```

Once it has the beginning of a block run, it scans forward on a new counter until it finds the end of that run or the row:

```
  until row[start].isGround
  local stop = start + 1
  while row[stop] and row[stop].isGround do
    stop = stop + 1
  end
end
```

Finally, it returns the start position as well as the function used to create the blocks in the right size and style, along with the width, obtained by subtracting the beginning from the end:

```
    stop = stop + 1
  end
  return start, row[start].isGround, stop - start
end
```

We here take advantage of a feature of Lua, where any non-nil value is considered true. So we can store functions with ground spaces that create the right sort of ground, and nil on those spaces that don't count as ground.

Creating the blocks

Back inside the loop, it's time to take the information from the iterator and make blocks with it. Multiply the index of the current row as well as the start and width of the block by the world's tile dimensions to determine the dimensions for the new floor:

```
        for start, kind, width in scanRow(row) do
          local block = kind(self, (start - 1) * self.HSize, (h - 1) *
self.VSize, width * self.HSize, self.VSize)
```

And finally we store the block in the list we created earlier:

```
          physics.addBody(block, 'static', {filter = categories.ground})
          table.insert(self.Blocks, block)
        end
```

What did we do?

We picked out individual streaks of consecutive blocks in each row of the tile map, and gave each one a physics identity, allowing it to block the passage of solid characters. We gave these blocks a generic grouping so that they will collide with nearly all other objects.

What else do I need to know?

There are two reasons to consolidate adjacent blocks rather than give each block tile its very own physics square:

- Box2D performs significantly faster when you give it fewer objects to handle, and the smaller number of blocks uses less memory as well
- Sometimes when another object scrapes along a series of stacked blocks, Box2D gets confused and makes it stop at one of the boundaries, like a person tripping over the seam in a sidewalk

Also, the iterator we created to scan for blocks is a bit inefficient, but fixing it (by returning the ends of blocks instead of their beginnings) makes the code's usage harder to understand and isn't worth it unless it has a significant impact on load times.

Making characters interact with the world

However, static blocks never interact with each other; they're only there to give characters and creeps something to interact with. Those objects also need to be modeled in the physics world in order for the physics model to be useful. We'll add physics modeling to the player character and monsters, so that they fall under gravity, land properly on the floor, and can't walk through walls.

Getting ready

Save `world.lua`, if necessary, and open `categories.lua` instead.

Getting on with it

Characters will need bodies that collide with the floors, but not with each other, to prevent collisions that transfer momentum.

Defining the character body category

In `categories.lua`, add another category under `ground`:

```
      maskBits = ID.sprite;
    },
    body = {
      groupIndex = ID.body;
      categoryBits = ID.sprite,
      maskBits = ID.wall;
    },
  }
```

Adding a body to characters

Save and move to the file `init.lua` in the `character` folder. This module takes a sprite and some other descriptive info and expands that sprite sheet into a complete game object with all its behaviors—except, at the moment, physics.

First we'll create an element description that is specifically solid, and uses the body filter we just defined, but otherwise inherits its shape info from the supplied body description.

```
return function(kind)
  local body = setmetatable({isSensor = false, filter = categories.
body}, {__index = kind.Form})
  return function(world)
    local self = display.newSprite(kind.Appearance, kind.Animations)
```

Now we'll add this shape as the base fixture in the sprite's new body:

```
   self.Mind = require "plan" (self.Brain, self, world, kind.
Personality(self))
    physics.addBody(self, 'dynamic', body)
    self.isFixedRotation = true
```

What did we do?

We've laid the framework here for all characters to interact with the ground, without interacting with each other. This eliminates a lot of issues like characters violently bouncing off each other.

What else do I need to know?

Box2D simulates bodies in three different ways. **Dynamic bodies** are fully simulated; they collide with all other objects (except when filters specify that they shouldn't), they move according to their own velocity, and their velocity is affected by gravity and other forces that you might apply. They bounce when two of them collide.

Static objects never move on their own. They typically represent ground, walls, or other immovable objects. You can move them by setting their x and y coordinates (outside of physics processing), but they never keep any velocity from one round to another. Box2D also optimizes collision detection against static objects in a way that gets much better performance if the objects are left standing still instead of being moved around a lot. Dynamic objects will bounce off of static objects, but never move the static object in the process, and static objects never create collision events with other static objects, no matter where you move them.

If you need an object that acts like a static object but will move regularly, make it a **kinematic object**. Like static objects, kinematic objects win any collision they're involved in, and are unaffected by gravity; unlike static objects, they can have a velocity that affects their movement over time. You cannot apply forces or impulses to a kinematic object and they pass through static objects and other kinematic objects without generating any collisions.

Responding to collisions with other characters

Even though we don't want characters colliding violently with each other, we still want them interacting when they come into contact. We'll deal with this by adding sensor elements (what Box2D calls **fixtures**) to our existing objects, with the same shapes as their solid portions; these sensors won't generate dynamic collisions, but they will trip collision events so that we can detect when they happen.

Getting ready

Save any other files and reopen `categories.lua`.

Getting on with it

Enemies shouldn't be concerned about colliding with each other, but collisions between enemies and the character are important.

Defining categories for enemies and the player

The player and enemy objects are very similar in their overall requirements; they collide with the same sorts of things, but they have different requirements as to what they need to ignore.

```
    maskBits = ID.wall;
  },
  player = {
    categoryBits = ID.spirit,
    maskBits = ID.spirit + ID.ladder;
  },
  enemy = {
    categoryBits = ID.spirit,
    maskBits = ID.spirit + ID.ladder;
  },
}
```

Each one belongs to a different group in order to control collisions, though.

```
  player = {
    groupIndex = ID.player;
    categoryBits = ID.spirit,
    maskBits = ID.spirit + ID.ladder;
  },
```

```
enemy = {
  groupIndex = ID.enemy;
  categoryBits = ID.spirit,
```

Applying categories to bodies

Save any open files and open `character/init.lua`. Find the module function, which accepts a character description and generates a character constructor function. This function is currently generating a solid body only for a new character:

Add a new body to the constructor function, based on the same template as the first one.

```
 local body = setmetatable({isSensor = false, filter = categories.
body}, {__index = kind.Form})
  local spirit = setmetatable({isSensor = true, filter = kind.Group},
{__index = kind.Form})
  return function(world)
```

Add this extra fixture to the physics body creation.

```
    self.Mind = require "plan" (self.Brain, self, world, kind.
Personality(self))
    physics.addBody(self, 'dynamic', body, spirit)
    self.isFixedRotation = true
```

Now that we have code to make use of the `Group` field of a character description, switch to `character/person.lua` and add this field to the character descriptions there. This file is already requiring the `categories` module.

```
humanoid{
  Name = "Hero";
  Appearance = "ranger_f";
  Personality = require "player";
  Group = categories.player;
  Speed = 64;
};
humanoid{
  Name = "Mummy";
  Appearance = "mummy";
  Personality = require "monster";
  Group = categories.enemy;
  Speed = 32;
};
```

Reacting to collision events

Now that each character has both a solid fixture to collide with walls, and a sensor fixture to detect collisions with each other, it's time to link some of that processing together.

Now that player and monster characters can collide with each other, we can resolve collisions based on what character is involved. Add some new options to the contact function in `character/init.lua`.

```
if other.Name == 'Ground' then
  self:dispatchEvent{name = 'Land'}
  self.Standing = other
elseif self.Allegiance == categories.player then
elseif self.Allegiance == categories.enemy then
end
```

If the player collides with something that is not ground, it's currently reasonable to assume that it must be an enemy, which hurts the player.

```
elseif self.Allegiance == categories.player then
  self:dispatchEvent{name = 'Hurt'}
elseif self.Allegiance == categories.enemy then
```

Make sure that collisions will trigger the `contact` function to process them, and the character's mind will be apprised of the fact that he or she has been hurt, in the body of the constructor.

```
self.isFixedRotation = true
self:addEventListener('collision', contact)
self:addEventListener('Hurt', self.Mind)
self:addEventListener('Fall', self.Mind)
```

We also need to specify that being hurt will pre-empt anything else the character is doing, and block their control for a moment. There is already a function to handle this, but we need to let this event interrupt other functions. Add a couple entries to the precedence table at the top of `character/init.lua`.

```
local precedence = {
  fall = {
    Land = 'idle',
    Hurt = 'flinch',
    Steer = 'fall',
```

Add the same line, `Hurt = 'flinch'`, to the walk and idle subtables.

What did we do?

By adding sensor components to the various character bodies, we can now trigger responses, such as reversing direction or reacting with damage.

What else do I need to know?

Sensor components are slightly limited compared to solid components, because there are no reactive collisions to disable or modify. The biggest indication of this is that they don't receive `preCollision` or `postCollision` events, only the collision event with both `began` and `ended` phases. This also means that their contact points can't be disabled, or have bounce or friction modified, because they don't generate any.

Bouncing off enemies as appropriate

Like any good platformer, our character will bounce off of enemies when he/she lands on top of them, but not when he/she hits them from the side. We'll accomplish this by checking the position of the collision on the main character.

Getting on with it

We've already created the basic behavior for when the character collides with an enemy, by taking damage and flinching back.

Checking for foot-head contact

Although there are a number of tests to determine whether a body has landed on top of another body, such as checking direction or comparing positions for horizontal overlap and vertical separation, we're going to create another sensor fixture. This one is intended to detect collisions with walls, rather than other sprites. Open `character/init.lua` and calculate the width of the new body in the main function:

```
local spirit = setmetatable({isSensor = true, filter = kind.Group},
{__index = kind.Form})
  local width, height = calculateBaseWidth(body)
  width, height = width or (sprite.width / 2 - 2), height or (sprite.
height / 2)
    return function(world)
```

Use this base width to create an outline along the bottom of the sprite:

```
  width, height = width or (sprite.width / 2 - 2), height or (sprite.
height / 2)
  local feet = {isSensor = true, filter = categories.body; shape =
{width, height; width, height + 2; -width, height + 2; -width, height}
}
    return function(world)
```

Include this fixture as the third component of each character body:

```
  local feet = {isSensor = true, filter = categories.body; shape =
{width, height; width, height + 2; -width, height + 2; -width, height}
}
  physics.addBody(sprite, 'dynamic', body, spirit, feet)
```

Modify the `contact` function so that its collision recognition checks for this specific overlap. This can be used to distinguish between landing on the ground or running into a wall:

```
    if other.Name == 'Ground' then
      if event.selfElement == 3 then
        self:dispatchEvent{name = 'Land'}
        self.Standing = other
      elseif self.Standing and self.Standing ~= other then
        self:dispatchEvent{name = 'Bump'}
      end
    elseif self.Allegiance == categories.player then
```

It can be used to distinguish whether the player character has landed on top of an enemy or bumped into it from the side:

```
    elseif self.Allegiance == categories.player then
      if other.Allegiance == categories.enemy and event.selfElement ~=
3 then
        self:dispatchEvent{name = 'Hurt'}
      end
    elseif self.Allegiance == categories.enemy then
```

It can also be used by the enemy to act more bouncy when the layer jumps on them. The player bouncing off the tops of enemies is part of their physics, not their control.

```
    elseif self.Allegiance == categories.enemy then
      if other.Allegiance == categories.player and event.otherElement
== 3 then
        local dX, dY = other:getLinearVelocity()
        other:setLinearVelocity(dX, -dY)
        self:dispatchEvent{name = 'Stomp'}
        timer.performWithDelay(350, function() self:removeSelf() end)
      end
    end
```

What did we do?

We distinguished between two different sorts of collision between the same pair of bodies, by using multiple sensors within a single body.

What else do I need to know?

Collision events also include x and y fields, which specify the point where the colliding bodies made contact. These also offer a way to check the particulars of a collision; however, at the time of this writing, there was a Corona bug affecting the coordinates reported in this event. If this seems like a simpler way to process your collision events, check the Corona update logs to see if this is now reporting reliably.

Controlling gravity to enable climbing

One of the benefits of the physics engine is that it frees us from handling gravity and its effects on falling and jumping in our own code; these are well-understood, well-established procedures, and coding them again ourselves makes no sense when there are already high-performance, well-tested tools available. However, as players of platform games, we frequently want to be able to reach certain high areas without having to jump up complicated staircases first. To create a ladder that we can both walk past and climb up, some special code is needed to interact with the gravity system; we'll practice detecting when the player is in a suitable place to change its physics (that is touching a ladder) by using a sensor object.

Getting on with it

There is one limitation of Corona's current implementation of Box2D that will oblige us to do a bit of work on our own. There is currently no way to ask what other objects a sensor or body is currently in contact with, so we have to remember objects when we start a collision with them, and then forget them when the collision is over.

Tracking contact with ladder regions

Open `character/init.lua` and add a table to store objects the character is in contact with:

```
self.Speed = kind.Speed
   self.Adjacent = {}
   self.Brain = enlighten(self)
```

In the collision processor, register any object in this table that starts a collision with the character.

```
local function touch(event)
  local self, other = event.target, event.other
  if event.phase == 'began' then
    self.Adjacent[other] = other
    if other.Name == 'Ground' then
```

When a collision ends, remove the departing object from the character's list.

```
  elseif event.phase == 'ended' then
    self.Adjacent[other] = nil
    if other.Climbable then
```

Detecting available ladders

When the player controller gets a climb input, it needs to check whether there is a ladder available. Open `game.lua` and add an adapting listener when the `Player` object is created:

```
UI:addEventListener('Release', self.Player.Mind)
UI:addEventListener('Vertical', self.Player)
function self.Player:Vertical(event)
  for object in pairs(self.Adjacent) do
    if object.Climbable then
      self:dispatchEvent{name = 'Climb'; unpack(event)}
      break
    end
  end
end
self.Focus = self.Player
```

Negating Gravity when Climbing

Return to `character/init.lua` and locate the `presence.climb` function inside the `enlighten` function. When the climbing process starts, reduce the effect of gravity on the character to nothing, to represent it holding on.

```
function presence.climb(direction)
  local action
  character.gravityScale = 0
  repeat
    local elapsed
```

Restoring gravity when releasing a ladder

When the player moves left or right while on a ladder, or jumps, the character releases the ladder and gravity reasserts its ugly dominance. At the end of `presence.climb`, restore the gravity scale to normal:

```
until action
    character.gravityScale = 1
    return action()
```

What did we do?

We allowed the game to interpret up-down control input only when in contact with a ladder, by keeping track of when we come into contact with a ladder, and controlling the effect of gravity on the character while climbing.

Selecting collisions by manipulating contacts

While the only thing that needs to be done when a sensor collides with another fixture is decide whether and how to react, **solid fixtures** (any fixture that isn't a sensor) have other accompanying behaviors; they don't **interpenetrate** (overlap), they can transfer **momentum** (bounce), and there can be friction when they collide. However, there is a hook in the library, extended by Corona, that allows us to use programmatic features to modify these aspects of a collision dynamically when it happens, so we'll control collisions by selectively disabling these contacts between objects on a case-by-case basis.

Getting on with it

When a collision event occurs, the event contains a reference to a `contact` object that can be used to examine or modify the characteristics of that specific collision.

Specializing the floor object

Start by creating a new file, `bridge.lua`. Open it and fill in the fact that it starts as a frontend to another module:

```
return function(parent, x, y, width, height)
  local self = require "rock" (parent, x, y, width, height)
  return self
end
```

Next, because the bridge appears only half as thick as a rock tile, adjust the height and y value passed to the existing module, making the body thinner and pushing it down the screen.

```
return function(parent, x, y, width, height)
  local trim = math.floor(height * 0.5)
  local self = require "rock" (parent, x, y + (height - trim), width,
trim)
  return self
end
```

Checking the direction

Our special floor will need another listener for imminent collisions:

```
local function testPosition(event)
end

return function(parent, x, y, width, height)
```

This listener needs to compare the position of the floor's top with the character's feet:

```
  local function testPosition(event)
  if event.target.y < event.other.y then
  end
end
```

Disabling the collision

In the event that the character is coming from the side or below, preventing the collision is simple:

```
  if event.target.y < event.other.y then
event.contact.enabled = false
  end
```

For any of this to actually work, we need to link the listener to the floor's `preCollision` events.

What did we do?

We created a class of platform that ignores collisions based on the direction the character was moving at the time, by using a `preCollision` handler to disable the contact.

What else do I need to know?

Pre-collision events can fire many times for two colliding events, and can start firing even before the objects' shapes are actually overlapping. This is because Box2D starts generating contacts for a pair of objects as soon as their bounding boxes (the smallest rectangle that still holds all of the object) overlap, even if the shapes contained within aren't actually touching. This is why contact objects have an `isTouching` property that should usually be checked when you respond to collision events.

Adding polish with custom fonts

Throughout the projects in this book up to this point, we've avoided using any fonts other than the default system font, whatever that might be, because the fonts that come installed on Android and iOS platforms are not consistent. However, fonts are a powerful element of a game or app's visual style and aesthetics; we can expand the options available to us by shipping our own fonts with our app. Corona makes this very simple. We'll review including the font data in your app, making it available to devices, and including it in your app.

Getting ready

Copy the file `Berenika-Bold.ttf` from the directory `original artwork/OFL 1.1/wmk69` in the project pack folder into the top level of your project directory.

Getting on with it

The Corona build process needs the font file to be in your project directory; after that, all you really need to do is refer to the font in your app.

Using a font in your app

To tell your app to use the font, all you typically need to do is refer to it by name. Open `scene.lua` and find the line near the top that reads as follows:

```
local bannerFont = native.systemFont
```

Change this line to read:

```
local bannerFont = "Berenika-Bold"
```

The lines that create the start and complete text use this variable to create their text objects, for instance:

```
local banner = display.newText(group, "Start!", 0, 0, bannerFont, 36)
```

Notice that the font name doesn't include the `.ttf` file extension.

Registering the font with iOS

No additional step is required to use a custom font on Android devices. For iOS, it's also required to include the font's name in the app's property list. Open the file `build.settings` and find the table named `plist`, defined inside a sub-table named `iphone` inside the table `settings`. Add a new line that lists the font file:

```
iphone = {
  plist = {
    UIAppFonts = {"Berenika-Bold.ttf",},
    UIStatusBarHidden = false,
```

If you use multiple custom fonts in your app, add all the filenames to the table on this line.

What did we do?

We made a new font available to our app, regardless of platform, and used it to render our own custom text objects. You can build your app for a device and see this custom text appear at the beginning of the game level.

What else do I need to know?

To also see your custom font appear in the simulator, you need to install the font in your operating system's usual location. On Mac OS, you can typically just double-click on the font file, which will open a display window in the **FontBook application**, including a button that says **Install Font**; click on this to load the font into your system. On Windows, follow the directions from the Microsoft Knowledge Base at `http://support.microsoft.com/kb/314960`.

On Mac OS and iOS devices, you can use either **True Type font** files (`.ttf`) or **Open Type font** files (`.otf`). On Windows and Android, only True Type fonts can be used.

Game over – wrapping it up

We've created an actively physics-driven game that takes advantage of many of Box2D's features: gravity (including gravity scale), collision filtering, using sensors to detect collisions without transferring momentum, and contact modification.

Can you take the HEAT? The Hotshot Challenge

Box 2D allows an object's `gravityScale` property to have any multiplier, not just 1 or 0. This means that you can use it to emulate buoyancy, or to simulate effects that Box2D doesn't recognize, like the wind resistance that distinguishes a falling leaf from a falling coin. Use this to emulate water (use a plain blue rectangle if needed, or supply a water sprite), so that the player sinks only slowly instead of falling, and the jump command works while the player is in the middle of the water.

Index

A

B

C

D

E

F

G

H

I

J

K

R

S

T

U

V

W

Thank you for buying
Corona SDK HOTSHOT

About Packt Publishing

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

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

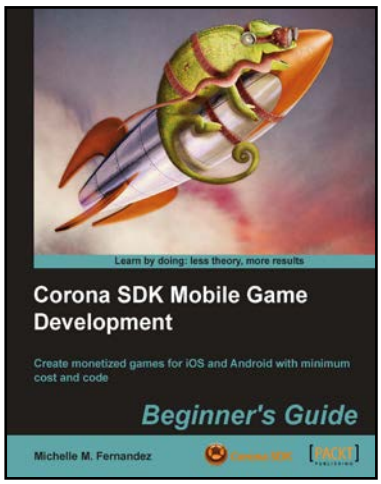
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

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

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

[PACKT]
PUBLISHING



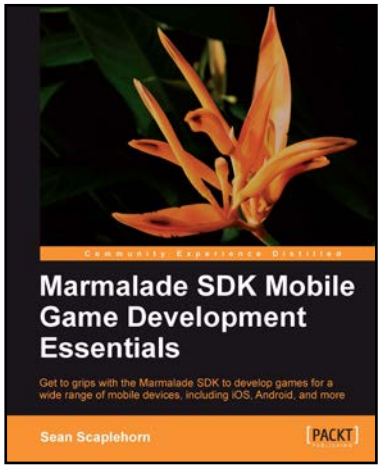
Corona SDK Mobile Game Development: Beginner's Guide

ISBN: 978-1-849691-88-8 Paperback: 408 pages

Create monetized games for iOS and Android with minimum cost and code

1. Build once and deploy your games to both iOS and Android
2. Create commercially successful games by applying several monetization techniques and tools
3. Create three fun games and integrate them with social networks such as Twitter and Facebook

---



Marmalade SDK Mobile Game Development Essentials

ISBN: 978-1-849693-36-3 Paperback: 318 pages

Get to grips with the Marmalade SDK to devekop games for a wide range of mobile devices, including iOS, Android, and more

1. Easy to follow with lots of tips, examples and diagrams, including a full game project that grows with each chapter
2. Build video games for all popular mobile platforms, from a single codebase, using your existing C++ coding knowledge
3. Master 2D and 3D graphics techniques, including animation of 3D models, to make great looking games

Please check **www.PacktPub.com** for information on our titles

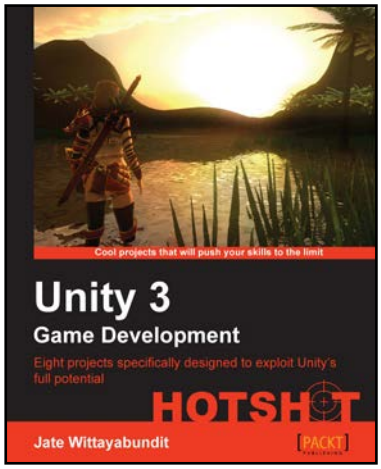
[PACKT] PUBLISHING

Building your First Mobile Game using XNA 4.0

ISBN: 978-1-849687-74-4 Paperback: 158 pages

A fast-paced, hands-on guide to building a 3D game for the Windows Phone 7 platform using XNA 4.0

1. Building a 3D game for the Windows Phone 7 platform
2. Drawing 2D and 3D graphics on Windows Phone.
3. Using the rich capabilities of the Windows Phone platform.
4. Creating a custom framework step by step that will act as a base for building (future) games.



Unity 3 Game Development Hotshot

ISBN: 978-1-849691-12-3 Paperback: 380 pages

Eight projects specifically designed to exploit Unity's full potential

1. Cool, fun, advanced aspects of Unity Game Development, from creating a rocket launcher to building your own destructible game world
2. Master advanced Unity techniques such as surface shader programming and AI programming
3. Full of coding samples, diagrams, tips and tricks to keep your code organized, and completed art assets with clear step-by-step examples and instructions

Please check **www.PacktPub.com** for information on our titles