Corona SDK Workshop

Darren Osadchuk, Ludicrous Software
www.ludicroussoftware.com
darren@ludicroussoftware.com
@DarrenOsadchuk

Table of Contents

0. Setting Up and Saying Hello

These instructions will help you set up the Corona SDK and a suitable IDE. There's also a simple "Hello World" project that you can try out to make sure everything is working.

Installing the Corona SDK

The first thing you'll need to do is install the Corona SDK. The SDK is free to download and use for development and testing. You will also be able to create device builds that you can test on your own device. When you're ready to ship your app, you'll need to become a subscriber so that you can create device builds for upload to your target marketplace.

Here are the steps to install the SDK:

1. To get the SDK, go to `www.anscamobile.com`, and click the "Download" button.

2. You'll be prompted to create a user account, which is required to download the SDK (it's free to create an account). Fill in your email address, select a password, and accept the Terms and Conditions, then click "Continue".

3. In the page that opens, select your operating system, and click the "Download" button to download the SDK installer.

4. After the download is complete, open the installer and follow the steps on-screen to install the SDK.

Installing an IDE

The IDE you use for development will differ depending on whether your computer runs OS X or Windows. I've included instructions for installing TextMate with the Corona SDK bundle if you're running OS X, or e if you're running Windows. You aren't required to use either of these, but can select any text editor that you may be comfortable with. I've included a short list of alternative choices for each operating system after the instructions for both (if you're a glutton for punishment you could use TextEdit or Notepad, although I wouldn't recommend it).

Installing the Corona SDK Bundle (OS X)

I use TextMate in OS X, which you can download from `www.macromates.com`. Here are the things you'll need to do to get it set up on your OS X machine.

Note: TextMate is not free software. If you don't already own it, there is a 30-day fully functional trial offered, so you can take it for a spin and see if you like it. If you decide to do this for the workshop, please don't install TextMate well in advance of the workshop, otherwise your trial may expire before the workshop actually happens!

After you've installed TextMate, I highly recommend that you install the Corona SDK TextMate bundle. It includes a number of handy snippets, code completion, and other features to make using Corona even easier than it already is. If you've never used TextMate before, you can watch a short video showing how to use the features in the bundle at `http://www.ludicroussoftware.com/corona-textmate-bundle/`.

The bundle is hosted on GitHub (`github.com/osadchuk/Corona-SDK.tmbundle`). You have two options for downloading/installing the bundle:

5. If you use Git for version control, then you can checkout the bundle into your `~/Library/Application Support/TextMate/Bundles` directory. Change into that directory and clone the bundle using

   ```
   git clone http://github.com/osadchuk/Corona-SDK.tmbundle.git
   ```

   You're good to go. The advantage of this method is the ease of getting new versions as the bundle is updated. If you want to specify a custom directory name for the bundle, just make sure it ends in ".tmbundle", otherwise TextMate won't recognize it.

6. If you don't want to use Git, then you can download the bundle as a zip file from GitHub. Here are the steps:

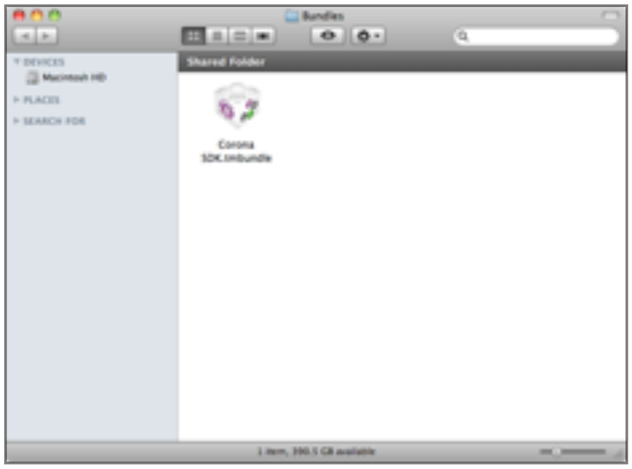
   a. Download the zip or tarball from GitHub and open it up.

   b. Rename the resulting directory to "Corona SDK.tmbundle", or whatever other name you prefer. Just make sure it ends in ".tmbundle".

   c. Copy the .tmbundle file to '~/Library/Application Support/TextMate/Bundles'.

   d. Restart TextMate or reload the bundles to see the Corona SDK Bundle under your bundles menu. When you open a Lua file or save a new file with a .lua extension, TextMate will automatically recognize the file as part of a Corona project and set the current bundle to the Corona SDK bundle.

Once the bundle is installed, you should see this in your Bundles directory:



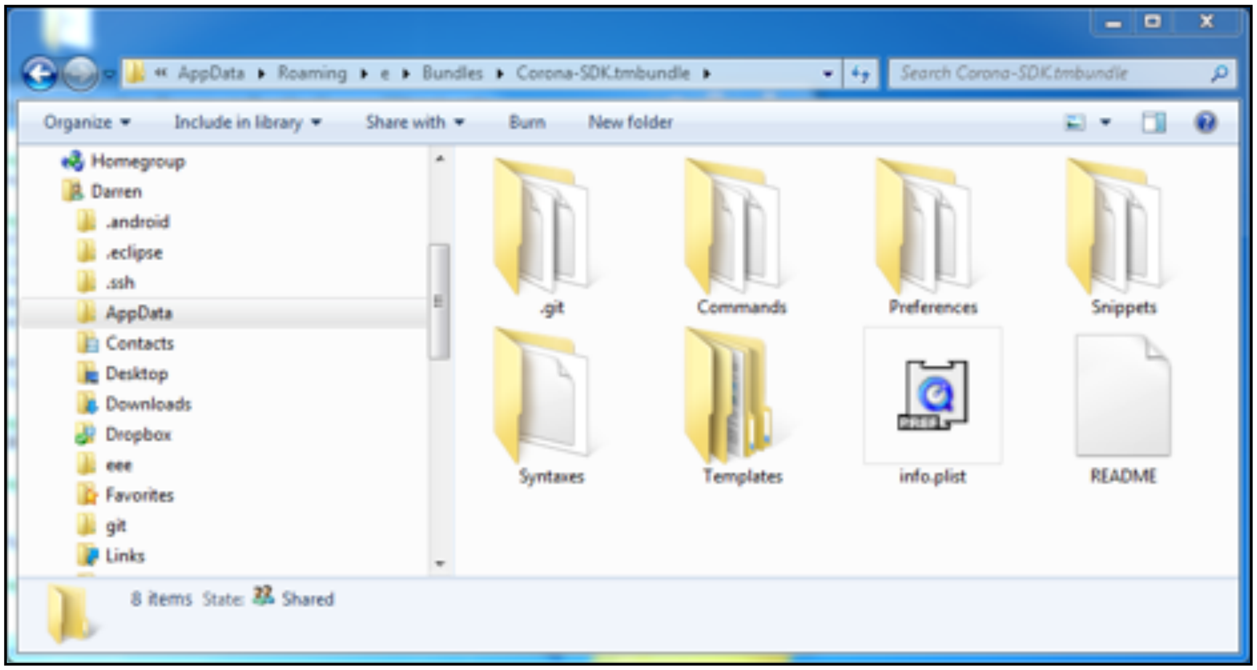
You should also see “Corona SDK” listed as an option under the “Bundles” menu in TextMate.

Installing e and the Corona SDK bundle (Windows)

E is a Windows-only text editor that has the benefit of supporting TextMate bundles. You can download e from http://www.e-texteditor.com/. Install it as you would any other Windows program.

Once E is installed, you can download and install the Corona TextMate bundle. The options for installation - using Git or downloading the zip file from GitHub - are the same for Windows as for OS X, but there are a couple of minor differences:

If you’re using Git, you need to clone the bundle into the `\Users\[username]\AppData\Roaming\e\Bundles` directory. If the directory doesn’t exist, create it and then use the ‘git clone’ command from the OS X instructions. You should end up with a directory called ‘Corona-SDK.tmbundle’ that contains the bundle contents.



If you download the zip file from GitHub, then extract the contents of the zip file into the `\Users\[username]\AppData\Roaming\e\Bundles\Corona-SDK.tmbundle` directory.

Regardless of the method you use, the end result should look like this:

Other IDEs

If you don't want to use TextMate or E, pretty much any other text editor will work just fine. Here are some other options to try, with varying levels of support for Lua.

OS X
- IntelliJ
- Sublime 2
- Xcode
- TextWrangler

Windows
- IntelliJ
- Sublime 2
- LuaEdit
- Notepad++

Hello World!

With the Corona SDK installed and your editor of choice ready to go, you can test your setup to make sure everything is working properly by creating a "Hello World" project.

Corona project folders are nothing more than folders containing all of the files required by your application, including the code files (with a `.lua` extension) as well as any graphics, audio, video, or other assets required. The most important file is `main.lua`. This is the entry point for your application, so this file must exist.

Here are the steps to say “Hello World” with Corona:

7. Create a new directory named `HelloWorld` for your project.

Note: if you’re using OS X, please make sure that there are absolutely no spaces in the path to your project. This includes the `HelloWorld` ***directory itself, as well as any parent directories. If there are any spaces anywhere in the path, the project will fail to load in the Corona Simulator.***

8. In your editor of choice, create a new file and save it with the filename `main.lua` in your project folder.

9. Enter the following code in `main.lua`:

```
local hello = display.newText(“Hello World!”, 50, 50, nil, 25)
hello:setTextColor(255, 255, 255)
print(“Hello World!”)
```

The `display.newText()` function requires five parameters, these are:

- The text to display
- The `x` and `y` position of the text, relative to the top left corner of the text box.
- The font to use; if the font is set to `nil` then Corona will use the default system font.
- The size of the font to use.

The `setTextColor()` function has three required parameters: the red, green, and blue values for the text colour you’d like to use. Each value ranges from 0 to 255. An optional fourth parameter sets the alpha value of the text (also on a range from 0 to 255).

The `print()` function will send whatever text you specify to the standard output, which is typically a terminal or command prompt window.

10. Save your work.

11. Start the Corona Simulator to test your work. Depending on your operating system and the editor you’re using, you have a few options for this:

    a. If you’re in OS X and using TextMate, you can use the ⌘-r keyboard shortcut to launch the Corona Simulator from within TextMate. This will start the Terminal application, which in turn will start the Simulator.

    b. If you’re in OS X but not using TextMate, you can launch /Applications/CoronaSDK/Corona Terminal. This will launch the Simulator for you. In the

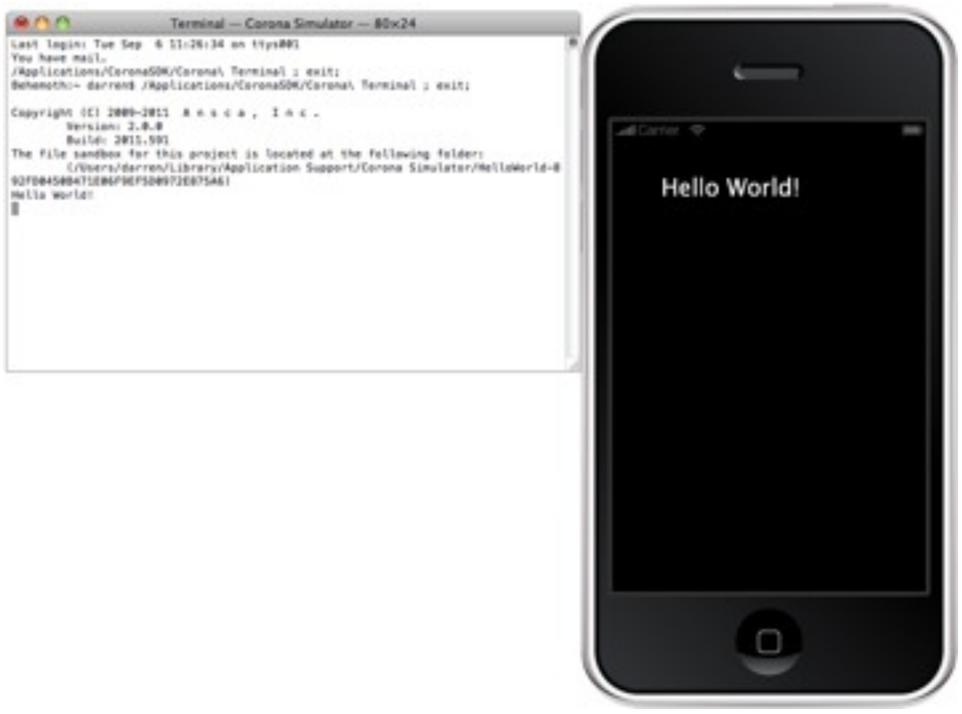
Simulator, select “Open...” from the “File” menu, and in the dialogue window navigate to your `HelloWorld` directory. You can also select the device to simulate if you don’t want to use the default iPhone skin. Click “Open”.

c. If you’re running Windows, then you need to open the Corona Simulator application, which by default is installed into \Program Files (x86)\Ansca\Corona SDK. This will open a command prompt window as well as the Simulator itself.

Note: In OS X, I recommend that you open the Corona Terminal instead of the Corona Simulator since the output of any print() statements or runtime errors will go to the standard output by default. If you run the Simulator directly, you won’t be able to see this information.

Additionally, the Corona Simulator can be set to automatically refresh whenever the project is modified. The option is available through the Preferences menu. I’d highly recommend enabling this option, since it makes testing and debugging go much quicker if you don’t have to manually refresh the Simulator whenever you make any changes to your project.

If everything worked, you should see the following in OS X or Windows, respectively:

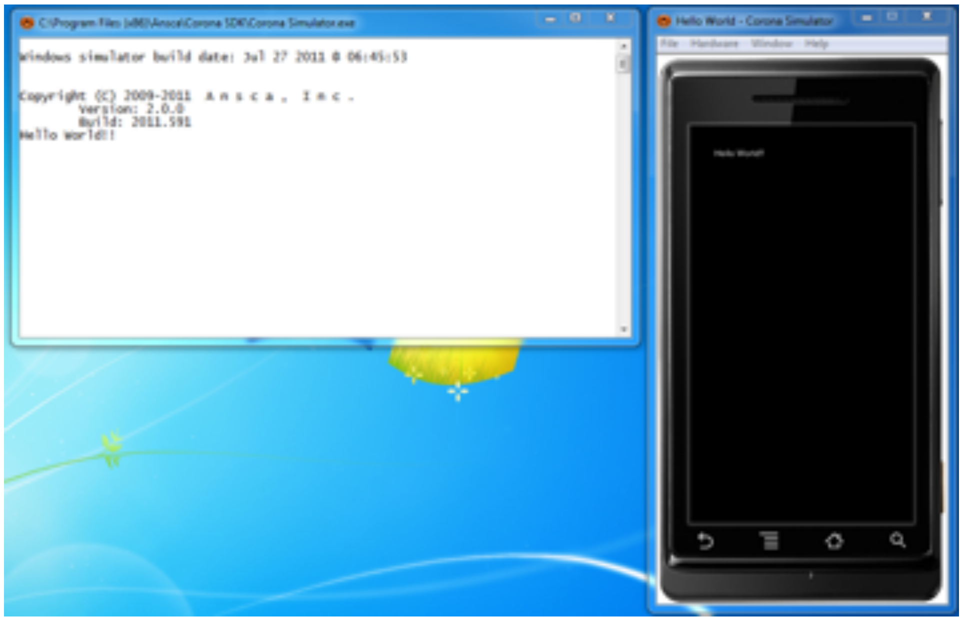


```
Last login: Tue Sep  6 11:26:34 on ttys001
You have mail.
/Applications/CoronaSDK/Corona\ Terminal ; exit;
Behemoth:~ darren$ /Applications/CoronaSDK/Corona\ Terminal ; exit;

Copyright (C) 2009-2011  A n s c a ,  I n c .
        Version: 2.0.0
        Build: 2011.591
The file sandbox for this project is located at the following folder:
        (/Users/darren/Library/Application Support/Corona Simulator/HelloWorld-8
92FD04508471E06F9EF5D0972E875A6)
Hello World!
```

Hello World!

Hello World in OS X


C:\Program Files (x86)\Ansca\Corona SDK\Corona Simulator.exe
Hello World - Corona Simulator
File
Hardware
Window
Help
Hello World!

```
Windows simulator build date: Jul 27 2011 @ 06:45:53


Copyright (C) 2009-2011  A n s c a ,  I n c .
        Version: 2.0.0
        Build: 2011.591
Hello World!!
```

Hello World in Windows 7

If everything went according to plan and you see something like one of these two screenshots, then congratulations! You're all set up and ready to start coding with Corona. If you see an error in your terminal window, check your code to make sure you followed the steps correctly.

1. Project Configuration

In this exercise you will configure the Corona project to suit the requirements of the Missile Command game.

Overview of the Game

The Missile Command game will have the following characteristics:

- The game will be played in landscape mode;
- A single code base will run on iOS and Android devices (phones and tablets);
- The game will leverage the Corona SDK's ability to scale content so that the appropriately sized graphics will be selected depending on device characteristics

Create a project folder

Create a new folder on your computer called "MissileCommand". This will contain all of the files for the game.

Make sure that there are no spaces in your folder names, and that there are no spaces in the path to your project folder. For example, "Missile Command" will cause problems, but "MissileCommand" is okay. Spaces in the path can cause problems for the Corona Simulator (at least in OS X).

build.settings

First, we'll create the `build.settings` file. This contains platform-specific information that the Corona SDK will use to create a device build.

1. Create a new file in your editor, and save it to the Project folder as "build.settings".

2. The information in `build.settings` is stored as a Lua table called `settings`. Add this table to the file:

```
settings = {
}
```

3. The `settings` table contains Lua tables nested within it. The first table to create will set the orientation of the game. MissileCommand will be played in landscape mode, and we will give the user the option of having their device in landscape mode in either direction. Add this table inside the settings table:

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

You may notice the "trailing comma" after the 'supported' table, and in later code snippets. Unlike in other languages, the trailing comma is legal in Lua.

4. Next, we will add the iOS-specific settings. Underneath the orientation table, create a new table called `iphone`. This table contains a `plist` table that can be used to set specific parameters for iOS devices (if you've done any Objective-C programming for iOS, then you'll recognize some of these settings from the `info.plist` file in Xcode projects). Our table will set the application version, the display name on the device, disable the 'gloss' effect on the icon, and set the application to exit completely when the application is suspended (due to an incoming call or if the user quits the game). Also, since we will be using a custom font in the game, we need to declare this for iOS. Add this code after the `orientation` table. Feel free to substitute your own reverse domain for the one used here:

```
iphone = {
      plist = {
            CFBundleShortVersionString = "1.0",
            CFBundleVersion = "1.0.0",
            CFBundleIdentifier = ↩
"com.ludicroussoftware.MissileCommand",
            CFBundleDisplayName = "MissileCommand",
            UIPrerenderedIcon = true,
            UIApplicationExitsOnSuspend = true,
            UIAppFonts =
            {
                  "ArmorPiercing.ttf"
            },
      },
},
```

5. Next, we can add the Android-specific information. This is broken up into two different tables. The first of these is called simply `android`, and all we'll do here is set the version code. Add this code after the `iphone` table:

```
android = {
      versionCode = "1"
},
```

6. Lastly, we'll create the `androidPermissions` table, which will contain all of the permissions our application may require (for example, if we were using GPS in the game, we would need to set that permission). We don't technically need to set any permissions for Android, but we'll include one for the sake of example. Add this table after the `android` table:

```
androidPermissions = {
      "android.permission.INTERNET"
},
```

Your build.settings file is now complete. Make sure to save the file before proceeding.

config.lua

`config.lua` will set up the initial configuration for the application. It consists of a single Lua table, called `application`, that tells Corona how to treat the application when it's run on different devices. For Missile Command, we will set the base size of the game to that of an iPhone 3 screen. We will also tell Corona how to scale the application when it's run on a device with different screen dimensions, and add information about the different resolution images that should be used on different devices.

7. Create a new file, saving it as "config.lua".

8. Add a new Lua table to the file, called `application`:

```
application = {
}
```

9. Inside the `application` table, add a new table called `content`, which will contain all of our settings:

```
application = {
      content = {
      }
}
```

10. Set the base width and height of our primary target device. For an iPhone 3, this is 320 x 480 - even though our game will run in landscape mode, we need to specify width/height in the default orientation of the target device. The settings in `build.settings` will set the orientation we want to use for the game. Add the code in bold:

```
content = {
      width = 320,
      height = 480,
}
```

11. Next, we want to tell Corona how to scale the application. We will set this to “letterbox”; other options are “zoomEven”, “zoomStretch”, and “none”. When letterbox scaling is used, Corona will scale the application up/down evenly until it completely fills either the width or the height of the device. If there is blank space on the sides or top of the screen, these will appear as vertical or horizontal black bars. This is similar to what happens when letterboxed movies are played on TVs that have different aspect ratios. However, unlike with TVs, in Corona we can place content inside this seemingly empty space, so that your application will fill the entire screen, regardless of size. Add the highlighted code:

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

12. Lastly, we need to add the list of image suffixes we wish to use. Corona will use dynamic image selection, so that the appropriate image is shown when the application is run on different devices. For example, the screen on an iPhone 3 is half the size of an iPhone 4’s screen. While we could let Corona use the iPhone 3 images and scale them up, the images will not look as crisp as possible. By specifying larger-scaled images for alternative devices, we can prevent this from happening. Add the code in bold to the content table:

```
content = {
     width = 640,
     height = 960,
     scale = "letterbox",
     imageSuffix =
     {
            ["@2x"] = 2,
     }
}
```

The format of the imageSuffix table is to set the image suffix used for each scaling increment that will have alternate images. In this case, we are only specifying one set of alternate images, for the iPhone 4 screen size of 640 x 960. Our images will have a suffix of “@2x”. Corona will make its best guess for other devices where the scale is slightly different from 2. We’ll see how this operates in practice later in the workshop.

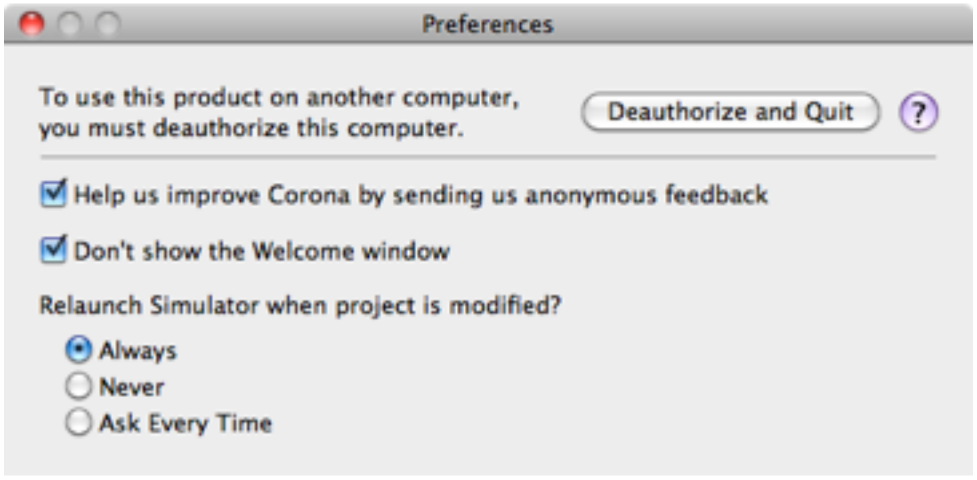
main.lua

Our two configuration files are finished and ready to go. The last thing we’ll do is create the `main.lua` file. Technically, this is the only file that is absolutely required in a Corona project, as it is the entry point into the application. So you could, if you wanted, code your entire application in `main.lua` (not that I’d recommend it, and not that we’ll be doing that in this workshop). To complete this exercise, you'll create the `main.lua` file and initialize some of the values required for proper cross-device performance.

13. Create a new file and save it as "main.lua".

At this point, you can try testing your project in the Corona Simulator. If you're using TextMate, the Command-r key combination will launch the project in the Simulator. First, the Terminal application will start up, and that will launch the Simulator. If you're using a different IDE, or are running Windows, you will have to manually start the Simulator and use the "open" command in the File menu to open the project.

Once the Simulator is running, you won't see very much, just the Simulator showing a blank screen. However, a nice feature of the Corona Simulator (at least in OS X) is that you can configure it to automatically reload the application whenever the project changes. This setting can be found in the application preferences window for Corona. Simply open that up and change the "Relaunch Simulator" setting:



To use this product on another computer, you must deauthorize this computer.

☑ Help us improve Corona by sending us anonymous feedback

☑ Don't show the Welcome window

Relaunch Simulator when project is modified?

● Always
○ Never
○ Ask Every Time

14. In main.lua, add the following line of code:

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

When you save the file, the Simulator should reload, and you should no longer see the status bar along the top of the screen.

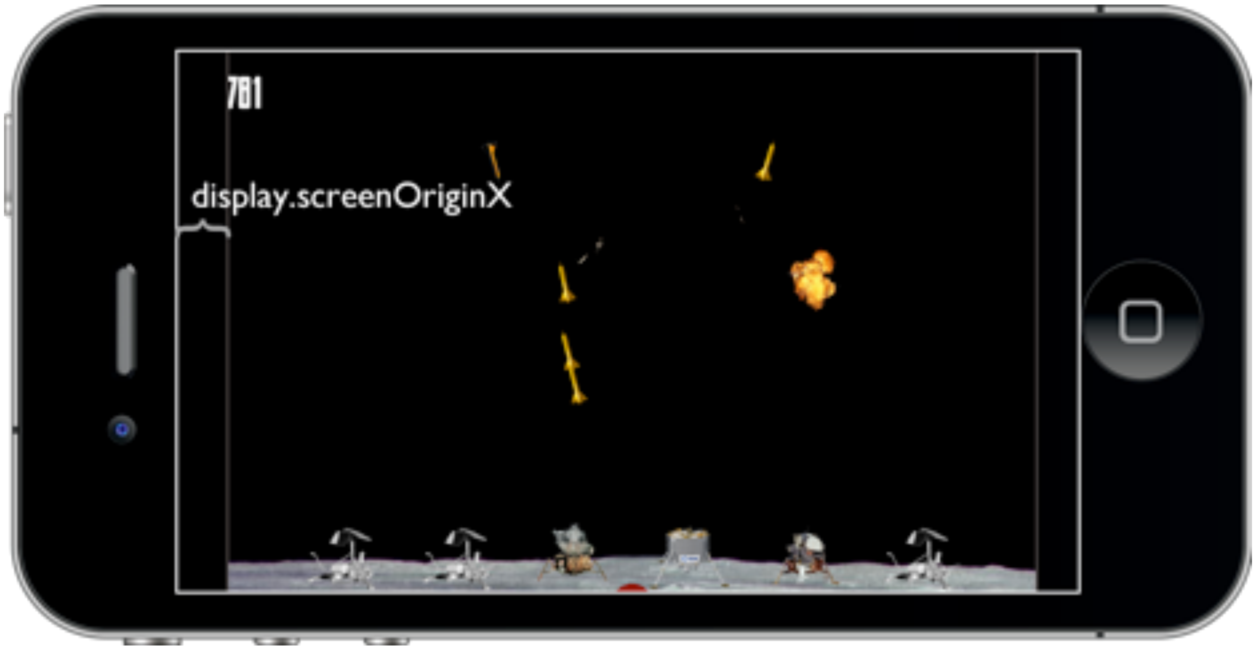
15. We'll also seed the random number generator, since our game will rely heavily on the random number function in Lua. Add this after the previous line of code:

```
math.randomseed(os.time())
```

16. Next, we'll configure some constants that will be used through the application. Rather than recalculate these values everywhere they're needed, the constants will be global variables. Add the following code to main.lua:

```
-- constants
STAGE_LEFT, STAGE_TOP = display.screenOriginX, display.screenOriginY
STAGE_WIDTH = display.contentWidth + STAGE_LEFT * -2
STAGE_HEIGHT = display.contentHeight + STAGE_TOP * -2
HALF_WIDTH = STAGE_WIDTH / 2
HALF_HEIGHT = STAGE_HEIGHT / 2
```

As mentioned earlier, when a Corona application is set to "letterbox" scaling, there may be extra space along either the top or side of the application if the aspect ratio between the application size and the device screen. The display.screenOriginX and display.screenOriginY properties let us find out how much offset there is between the position of the scaled content and the actual top or left position of the device, as can be seen in this image where the game screen, running on an iPhone 4, is scaled to show how it would appear on a Nexus One screen (represented by the white rectangle):



The screen origin properties will be negative numbers, so when calculating the actual width and height of the stage, we need to multiply these values by -2. This accounts for the letterboxing on both sides of the content, and the fact that the values are negative. This gives the actual width and height of the stage.

Lua does not have true constants, so these values can be changed if you're not careful. I've adopted the convention from other programming languages of using all capital letters when naming constants, which helps remind me not to do anything with them later in the program.

2. Dropping Bombs

In this exercise we'll start working on the game itself by preparing to get some bombs dropping on the moon.

Rather than code the entire application in `main.lua`, we'll use Lua's "modules" capability to break up the code into separate files. We'll create separate modules for the game and for most of the display objects that will be featured in the game.

Creating the Game Module

1. Create a new file, and save it as "Game.lua".

2. A module defines a Lua table, which is returned by the module when it is added to the application via the `require()` function. So, let's start by setting up the table that will get returned. In subsequent steps we'll make this table actually do stuff. Add the following code to `main.lua`:

   ```
   local Game = {}

   return Game
   ```

3. Next, we'll add a function called `new()` to our table, which will be called when we want to actually create the game. This function will return a display group that contains the game objects and game logic. To start, we'll simply define a couple of variables that we'll need to get started. To keep the code organized, we'll break up our declarations into logical groups, as indicated by the comments. Add this code before the `return` statement:

   ```
   function Game.new()

        local game = display.newGroup()
        game.id = "game"

        -- display objects
        local sky, ground

        return game

   end
   ```

4. Before we integrate this into `main.lua`, let's actually get something showing up on the screen. We'll start with the background, which in this case is the sky and an

image of the moon. Let's declare the function that we'll need; add the highlighted code immediately before the `return game` statement:

```
-- functions
local initGround

return game
```

5. Copy the "Ground.png" image from the supplied assets into your project folder.

6. Next, let's actually write the `initGround()` function. This function will add the Ground.png image to the game display group, and then position it along the bottom edge of the screen (all positioning is done relative to the centre of the image, which is the default behaviour for Corona). Add this code immediately before the "return game" statement:

```
function initGround()
      sky = display.newRect( game, _G.STAGE_LEFT, _G.STAGE_TOP, ↩
_G.STAGE_WIDTH, _G.STAGE_HEIGHT )
      sky:setFillColor(25, 25, 55)

      ground = display.newImageRect(game, "Ground.png", 600, 86)
      ground.x = _G.HALF_WIDTH
      ground.y = _G.STAGE_TOP + _G.STAGE_HEIGHT
end
```

There's a little trick at work here. By using the display.newImageRect() function, we're invoking Corona's ability to dynamically select an image based on our device characteristics, as we'd set it up in config.lua earlier. However, in this specific case, we're only supplying a single image, which is actually properly sized for an iPhone 3, rather than an iPhone 4. Since it's a less important background image we can take some texture memory.

7. The last thing we'll do before we integrate this into main.lua is create a public function that will do all of the initialization for us. Add this function after the initGround() function:

```
function game:startGame()
      initGround()
end
```

One difference you'll note is that `initGround()` is a local function, while `startGame()` is a property of the game display group. This makes it a public function.

Integrating the Game Module

Next, we'll actually set up `main.lua` so that it will add the Game module and start the game (such as it is right now).

8. In main.lua, add the following code after the constants:

```
-- libraries
local Game = require("Game")

-- display objects
local game

game = Game.new()
game:startGame()
```

This code does a few things. First, it uses Lua's `require()` function to add the Game module, storing the value returned by the module in the local Game variable. This means that Game is storing a reference to the `Game` table defined in Game.lua. Next, it declares a "game" display object, and then assigns the value returned by `Game.new()`, which is the game display group, to the game variable. Lastly, it calls the `startGame()` function.

If you test the game now, you should see the image of the moon along the bottom edge of the screen in the Corona Simulator! You can rotate the device using the menu options in the Simulator to see how Corona handles different orientations, and switch device skins to see how the ground looks on different devices.

The Game Loop

The game itself will meet the following requirements:

- The Corona physics API will be used, but only for collision detection;
- A main run loop will cycle through all of the game objects that need to be moved (essentially, bombs that drop and player missiles that are fired) and update their position.
- The run loop will also determine whether it is time to drop another bomb.
- The game will not give the appearance of having discrete levels, but we'll use the level concept to increase the difficulty as the game progresses.

Rather than simply drop another bomb at a regular interval, which would be somewhat predictable and rather boring, we'll increase the pace at which bombs fall over the course of a "level".

9. First, we'll tell Corona that we want to use the physics API in the game. We'll include the physics module and start it. Then, we'll turn off gravity, since we're going to manually control our physics objects. We'll also set the draw mode of the physics engine to "hybrid", which shows an overlay of physics bodies on top of regular display objects. Add the following lines of code near the top of `Game.new()` in `Game.lua`:

```
-- enable physics
physics = require("physics")
physics.start()
physics.setGravity(0, 0)
physics.setDrawMode("hybrid")
```

10. Next, declare the functions we'll require to set up a level, start the game running, and handle bomb dropping and object moving. Add the code in bold to the function declarations:

```
-- functions
local initGround
local configureLevel, startListeners, run
local checkForBombDrop, dropBomb, moveObjects
```

11. Stub in all the functions we'll require for this by adding the code in bold:

```
function configureLevel()
end

function startListeners()
end

function run()
end

function checkForBombDrop()
end

function dropBomb()
end

function moveObjects()
end

function game.startGame()
     initGround()
     configureLevel()
     startListeners()
end
```

Setting up Levels

The levels in the game will be more conceptual than actual levels (as in games like Angry Birds). A level will consist of a number of bombs that are dropped over a certain time period. The time period will be defined in frames. As the game progresses, the difficulty will be increased by simultaneously increasing the number of bombs in a level, decreasing the span of time during which they will fall, and increasing the speed at which they fall.

12. Declare the variables required to determine when bombs should be dropped. In the first level, 5 bombs will fall over the course of 600 frames. Above the function declarations, add the following block of code:

    ```
    -- variables
    local frameCounter, bombSpeed, bombDropTimes
    local bombsThisLevel = 5
    local levelDuration = 600
    local currentLevel = 0
    ```

13. We'll also set a constant for the base bomb speed, which will act as a minimum speed that we can use in our calculations to determine the actual bomb speed for a particular level. Add this after the variable declarations:

    ```
    -- constants
    local BASE_DROP_TIME = 400
    ```

14. To set up our levels so that the bombs fall less frequently to start, but increase in intensity closer to the end of a level, we'll use the `math.sin()` function to determine when the bombs should fall, and store those times in a table called `bombDropTimes`. Add the following code to the `configureLevel()` function:

    ```
    function configureLevel()
         bombsThisLevel = bombsThisLevel + 2
         levelDuration = levelDuration - 10
         if levelDuration < 200 then
               levelDuration = 200
         end
         bombDropTimes = {}
         local interval = 1 / bombsThisLevel
         for i = interval, 1, interval do
               local dropTime = math.round(math.sin(i * (math.pi / 2)) * ↩
    levelDuration)
               bombDropTimes[#bombDropTimes + 1] = dropTime
         end
         bombSpeed = BASE_DROP_TIME - currentLevel * 10
         currentLevel = currentLevel + 1
         frameCounter = 0
    end
    ```

15. Next we'll create an event listener that will listen for the “enterFrame” event and fire an event handler function whenever the event occurs (events in Corona are similar to those in other programming languages such as ActionScript 3). The event handler will be responsible for moving objects around the screen. Add the following line of code to `startListeners()`:

```
function startListeners()
      Runtime:addEventListener("enterFrame", run)
end
```

16. Now that we have our event handler firing on every frame, we need to tell it to do something. The first thing we'll do is move any objects that need to be moved. Then, we'll increment the `frameCounter` variable, which we'll use to check against the values in `bombDropTimes`. Every time there's a match, it's time to drop another bomb. Add the code in bold to the `run()` function:

```
function run()
      moveObjects()
      frameCounter = frameCounter + 1
      checkForBombDrop()
end
```

17. With `frameCounter` incrementing on every frame, we can determine when it's time to drop a bomb. Add this code to `checkForBombDrop()` and `dropBomb()` - for now we'll just `print()` something to make sure everything's working:

```
function checkForBombDrop()
      if #bombDropTimes == 0 then
            return
      end
      if frameCounter == bombDropTimes[1] then
            table.remove(bombDropTimes, 1)
            dropBomb()
      end
end

function dropBomb()
      print("dropBomb")
end
```

This code does a few things, so let's look at each in turn:

a. First, we check to see if the length of the bombDropTimes table is equal to 0 - in Lua `#tableName` returns the length of an array (similar to `Array.length` in AS3 or JavaScript). If it is, we stop the execution of the function immediately, as we know for sure that there are no further bombs to drop in this level.

b. Next we check the current value of `frameCounter` against the value of the first element in the `bombDropTimes` array. If they match, then we know it's time to drop a bomb. The element is removed from the array as it's no longer needed, and the `dropBomb()` function is called.

c. Lastly, we print out a statement showing that a bomb should be dropped.

Now, let's actually create a bomb to drop.

3. Bombs to Drop

In this exercise we'll create the Bomb module, which will contain the logic for the enemy bombs that will rain down on the moon.

Bomb.lua

1. Create a new file and save it as "Bomb.lua".

2. Copy the images "Bomb.png" and "Bomb@2x.png" from the supplied assets into your project folder.

3. Add the following code. We're not really returning anything in `new()` just yet (`bomb` is declared but not initialized, so its value is `nil`), but we'll take care of that soon:

```
local Bomb = {}

function Bomb.new()
     local bomb

     return bomb
end

return Bomb
```

4. We're going to create a constant that we can use when we need to convert between degrees and radians. Since display object rotation is expressed in degrees, but the math functions deal in radians, we'll need to convert between the two. Although Lua has built-in functions that can do this (`math.deg()` and `math.rad()`), they're quite a bit slower than a "manual" calculation. Add this after the Bomb table is declared:

```
local RADIANS_TO_DEGREES = 180 / math.pi
```

5. Next, we're going to add a parameter to our `Bomb.new()` function, which will be the `x` value that the bomb should target. This value will be determined by the game module, and passed to the bomb when it's created - in effect, the game will tell the bomb where to go. We're also going to use the Lua `assert()` function to make sure that the parameter has been passed in; if it's not, the game will throw an error. Add the code in bold:

```
function _M.new(targetX)

     assert(targetX, "Required parameter missing")
```

6. Now, we can actually create the bomb display object. We'll use the `display.newImageRect()` function, which will instruct Corona to dynamically substitute our higher-resolution image where required. We will also declare a few variables that will be used later on to move the bomb. `deltaX` and `deltaY`, as the names imply, will store numbers that will tell the bomb how much it should move along the x and y axes, respectively, in each frame. `targetX` and `targetY` store the bomb's target coordinates, and `bombX` and `bombY` store the bomb's x and y position. For the latter two variables, we store these in local variables because it is faster in Lua to manipulate local variables than to look up values (such as the `x` and `y` properties) in a table. Add this after the `assert()` call:

```
local bomb = display.newImageRect("Bomb.png", 25, 10)
bomb.id = "bomb"
bomb.x = -1000

local deltaX, deltaY
local targetY, bombX, bombY
```

7. Next, we can turn the bomb into a physics body, so that we can use the Box2D collision detection to find out when a bomb has hit something. To do this, add the following code after the variable declarations:

```
    physics.addBody(bomb, "dynamic", {isSensor = true, ↩
shape = BOMB_SHAPE})
```

8. As you can see, the bomb uses a custom physics shape, `BOMB_SHAPE`, instead of the default rectangle that is the same size as the image. A custom shape is defined as a Lua table that consists of a series of x/y coordinates, relative to the centre point of the image and moving in a clockwise fashion. Add the shape data required for the bomb after the other constants:

```
local BOMB_SHAPE = {-6,-1, 8,-1, 9,-1, 8,1, -6,1, -11,4, -11,-5}
```

Lastly, we need to add the three public functions required: one that is called when a bomb is first created and fired, one that is called on every frame to move the bomb, and one that is used to remove the bomb from the game after it has hit a target

9. We'll start with `bomb:fire()`. This function takes one parameter, which is the amount of time the bomb should take to drop to the ground. It will be passed in from the game module when a bomb is instantiated:

```
function bomb:fire(dropTime)
    bombX = math.random(_G.STAGE_WIDTH)
    bombY = -100
    targetY = _G.STAGE_HEIGHT
    local diffX = targetX - bombX
    local diffY = targetY - bombY
    deltaY = diffY / dropTime
    deltaX = diffX / dropTime
    local targetAngle = math.atan2(diffY, diffX) * RADIANS_TO_DEGREES
```

```
          bomb.rotation = targetAngle
     end
```

Let's look at each step in turn:

a. A starting `x` position is selected at random, and the bomb is positioned offscreen, just above the top of the stage.

b. Next, its `targetY` property is set to be equal to the height of the stage, meaning the bomb will aim for the ground.

c. Now that we have a starting `x`/`y` position and a target `x`/`y` position (recall that `targetX` is set when the bomb is instantiated), we can calculate the `deltaX` and `deltaY` values. This is done by determining how far the bomb needs to move along each axis (`diffX` and `diffY`), and dividing that value by the time in which it has to move that far.

d. Lastly, we rotate the bomb using the value returned by `math.atan2()` so that it is pointing at its target destination.

10. Next, we'll create the `bomb:move()` function. This function simply adds the values for `deltaX` and `deltaY`, calculated by `bomb:fire()`, to the bomb's `x`/`y` properties:

```
function bomb:move()
      bomb.x = bombX
      bomb.y = bombY
      bombX = bombX + deltaX
      bombY = bombY + deltaY
end
```

11. Lastly, we'll clean up after ourselves when the object is removed by the game module. We need to remove the event listener that will have been added to the bomb - we haven't added this yet, so don't worry about it right now. We'll come back to it in the next exercise. We remove the bomb from the display list, and lastly, we set its value to `nil`, which marks it for garbage collection:

```
function bomb:destroy()
      bomb:removeEventListener("collision", bomb)
      bomb:removeSelf()
      bomb = nil
end
```

4. Actual Bombs Dropping

In exercise 2, we got some conceptual bombs dropping via the unexciting `print()` statement. In exercise 3, we set up a bomb “class”, ready and waiting to start dropping. In this exercise, we’ll connect the two and get some actual bombs dropping on the moon.

Dropping Bombs

In theory, getting the bombs to drop requires two steps: first, when it’s time to drop a bomb, we need to instantiate a bomb instance. Second, we need to make sure that our run loop is moving the bombs. In practice, there are a few other things we’ll need to set up. Recall that we’re using the Corona physics engine to handle collision detection, so we’ll need to give the bombs something to collide with, and listen for those collisions. In the previous chapter we set up the Bomb module so that the `targetX` parameter is required, so we also need to give the bombs something to aim. Later on, we’ll give the bombs some proper targets, but for the time being we’ll just pass in a random number so that we can make sure things are working properly so far.

1. Before we can make use of the Bomb module, we need to `require()` it in the Game module. Near the top of the `new()` method, add the following code - I placed mine immediately after the physics engine initialization:

    ```
    -- modules
    local Bomb = require("Bomb")
    ```

2. First, we need someplace to put the bombs. To make it easy to move all of the bombs during the run loop, we’re going to put them into the `bombs` display group.That way, we can iterate through the display group’s child objects and update their positions. Declare the display group underneath the display object declarations by adding the code in bold:

    ```
    -- display objects
    local sky, ground

    -- display groups
    local bombs
    ```

3. Now we need to instantiate `bombs`. We’ll also add it to our main `game` display group. We need to be careful about how/when we insert `bombs` into `game`, since we want the

bombs to always appear on top of everything else. So, we'll create and insert it after the ground has been created, in the `startGame()` function:

```
function game:startGame()
     initGround()
     bombs = display.newGroup()
     game:insert(bombs)
     configureLevel()
     startListeners()
end
```

4. Next, find the `dropBomb()` function in `Game.lua`. To drop a bomb, we need to select a target, instantiate the bomb, add it to `bombs`, and then call the `fire()` function. Replace the `print()` statement with the following code:

```
local targetX = math.random(0, _G.STAGE_WIDTH) + _G.STAGE_LEFT
local bomb = Bomb.new(targetX)
bombs:insert(bomb)
bomb:fire(bombSpeed)
```

5. Next, find the `moveObjects()` function. Here, we need to iterate through the `bombs` display group so that we can update all of the objects it contains. We do this by calling the `Bomb.move()` function we set up in the previous exercise. When iterating through a display group, we can use the `numChildren` property to find out how many display objects it contains, and access those objects using array notation (similar to iterating through an array in Javascript/AS3).

```
function moveObjects()
     for i = bombs.numChildren, 1, -1 do
           local bomb = bombs[i]
           bomb:move()
     end
end
```

Hitting the Moon

Test out the project so far. After a brief delay, you should start to see some bombs falling on the moon. Enjoy the show for a moment - and notice that the bombs just keep on falling, and falling, and falling. Obviously, this is a bit of a problem, so we need to give them something to hit, and we need to clean up after them when they've hit something.

To do this, we'll add a rectangle to the screen that will act as the ground for the purpose of the physics engine. Once that's in place, we also need to listen for collisions between physics objects, and take the necessary action when they happen.

6. Declare a new display object, which we'll call `groundPhysics`:

```
-- display objects
local sky, ground, groundPhysics
```

7. In the `initGround()` function, add the code shown in bold. This creates the vector rectangle, and then uses the `physics.addBody()` function to add it to the physics engine. Additionally, we assign an `id` property to the ground, which we'll use during collision detection to find out what the bomb has collided with (the bomb can collide with the ground, player missiles, or the targets on the ground):

```
function initGround()
     ground = display.newImageRect( game, "Ground.png", 600, 86 )
     ground.x = _G.STAGE_WIDTH / 2
     ground.y = _G.STAGE_HEIGHT - ground.height / 2
           groundPhysics = display.newRect(game, ↩
_G.STAGE_LEFT,_G.STAGE_HEIGHT + _G.STAGE_TOP - 2, _G.STAGE_WIDTH, 2)
     groundPhysics.isVisible = false
     physics.addBody(groundPhysics, "static")
     groundPhysics.id = "ground"
end
```

8. Return to the `dropBomb()` function. We need to add the "collision" event listener. We'll do this by creating a "table listener". To do this, we need to create a property on the table that has the same name as the event we're listening for. The property will be a reference to the event handler function Since we want the bomb we're creating to listen for the "collision" event, we need to point `bomb.collision` to our event handler. Then, when we add the event listener, instead of passing in a function, we pass in the table that has the listener. All display objects in Lua are essentially tables, so we can add the listener directly to the bomb. Add the code shown in bold:

```
function dropBomb()
     local targetX = math.random(0, _G.STAGE_WIDTH)
     local bomb = Bomb.new(targetX)
     bombs:insert(bomb)
     bomb.fire(bombSpeed)
     bomb.collision = collisionHandler
     bomb:addEventListener("collision", bomb)
end
```

9. Now we need to create the `collisionHandler()` function, so that we can take the appropriate action when the bomb hits something. First, we need to declare the function, so add `collisionHandler` to our list of declared functions:

```
-- functions
local initGround
local configureLevel, startListeners, run
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
```

10. Next, let's define the `collisionHandler` function itself. Add the function below `moveObjects()`. Because we're using a table listener, Corona will pass two objects to the event handler, `self`, which in this case is a reference to the bomb that collided

with another object, and `event`, an object that contains information about the collision. Collision events in Corona have different "phases", so we'll listen for a collision only when it actually begins:

```
function collisionHandler(self, event)
     if event.phase == "began" then
          print(self, "collided with", event.other.id)
     end
end
```

11. When the bomb hits the ground, an explosion will occur and the bomb will be removed from the game. Similarly, different actions will occur when the bomb hits other objects, so we need to add an `if` statement to take the appropriate action depending on the object hit. We can determine this by looking at the `id` property. We'll add the explosion in the next exercise, so for now we'll just remove the bomb. Modify `collisionHandler()` by removing the print statement and adding the code shown in bold:

```
function collisionHandler(self, event)
     if event.phase == "began" then
          local objectHit = event.other
          local id = objectHit.id
          if id == "ground" then
               groundHit(self)
          end
     end
end
```

12. Next, declare the `groundHit` function:

```
-- functions
local initGround
local configureLevel, startListeners, run
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit
```

13. Add `groundHit()` below `collisionHandler()`. Later on, it will be responsible for instantiating the ground explosion animation as well, but for now it will simply make sure that the bomb is removed:

```
function groundHit(bomb)
     bomb:destroy()
end
```

Now, when you test the project, the bombs should fall and then vanish from the screen when they hit the ground.

One Last Tweak: More Bombs

While we're dealing with collisions, let's modify the game so that it continues to drop bombs. We don't have discrete levels in the game, but we do want to increase the difficulty as the game goes on. We do this by calling `configureLevel()` again.

14. To get more bombs dropping, we'll wait until the last bomb in the current level has collided with something. By waiting until this point, we don't run the risk of having our levels "overlap" as the game goes on (which, as I found out during development of the game, creates quite a few bombs on the screen at one time!). We know that a bomb has collided with something when our collision event handler function is called. We know that the bomb that collided is the last bomb if the `bombDropTimes` table has no more entries in it (i.e. has a length of 0). So, to make this work, add the code in bold to `collisionHandler()`:

```
function collisionHandler(self, event)
     if event.phase == "began" then
           local objectHit = event.other
           local id = objectHit.id
           if id == "ground" then
                 groundHit(self)
           end
           if #bombDropTimes == 0 then
                 configureLevel()
           end
     end
end
```

Now that our bombs are dropping and we know when they've hit the ground, in the next exercise we'll start shooting back.

5. Shooting Back

Now, let's add the ability to fire at incoming bombs and earn points for destroying them. Missiles will be fired at whatever location on the screen the player touches. Implementing this will require a few steps:

- create a listener for player input;
- create a new missile and fire it at the location touched by the player;
- listen for collisions between bombs and missiles;
- keep score.

Before we tackle that, we'll create the missile module.

Something to Fire

Here are the features of the Missile module:

- We will pass in parameters telling the missile where to start and where to go;
- Like the Bomb, the Missile has a `move()` function that will update the missile's position on every frame. Unlike the Bomb's `move()` function, the missile will return a boolean value telling the game's `moveObjects()` function whether it's reached its destination. In other words, bombs will not simply keep going until they fly off the screen; they will fly to the position on the screen touched by the player, and then they will explode.
- The missile also has a `destroy()` function, which is called when the missile explodes.

In addition to these requirements, we need do some calculations to orient the missile in the right direction and figure out how far it should move in each frame.

1. Copy the "Missile.png" and "Missile@2x.png" images into your project directory.

2. Create a new file and save it as `Missile.lua`. We'll add in some local constants and stub in the `new()` function. The `MISSILE_SHAPE` constant will be used when we create a physics body out of the missile; the `BASE_SPEED` constant is the speed of the missile measured in pixels per frame. Finally, we'll verify that all of the required parameters are present when `new()` is called:

```
local Missile = {}

local MISSILE_SHAPE = {-5,-1, 9,-1, 10,-1, 9,1, -5,1, -11,5, -11,-5}
local BASE_SPEED = 4
local RADIANS_TO_DEGREES = 180 / math.pi

function Missile.new(targetX, targetY, originX, originY)

     assert(targetX or targetY or originX or originY, ↩
"Required parameter missing")

end

return Missile
```

3. Next, we can create a missile from our sprite set, set its scale as required, and return it:

```
assert(targetX or targetY or originX or originY, "Required ↩
parameter missing")

local missile = display.newImageRect("Missile.png", 25, 10)
missile.id = "missile"

return missile
```

4. We'll declare a couple of variables, `deltaX` and `deltaY`, that will store the result of our calculations to determine how far to move the missile on each frame. We'll also create the physics body using the custom shape we'd defined previously. Add this code before the `return missile` statement:

```
local deltaX, deltaY

physics.addBody(missile, "static", {isSensor = true, ↩
shape = MISSILE_SHAPE})
```

5. Next, we'll create a local `fire()` function that will perform the trigonometric calculations required to determine the missile's `deltaX` and `deltaY` values and point it toward its target. Add this code before the return statement:

```
local function fire()
     missile.x, missile.y = originX, originY
     local diffX = originX - targetX
     local diffY = originY - targetY
     missile.rotation = math.atan2(diffY, diffX) * ↩
  RADIANS_TO_DEGREES + 180

     local distance = math.sqrt(diffX * diffX + diffY * diffY)
     local theta = math.asin(diffX/distance)
     deltaX = BASE_SPEED * math.sin(theta)
     deltaY = BASE_SPEED * math.cos(theta)
end

fire()
```

6. Now we can create the two public functions. The first is the `move()` function that will update the missile's position and determine if it's reached its target. To move the missile, all we need to do is subtract the delta values from the missile's current position. Then we check its distance from the target. If the distance is less than 5 pixels along both the `x` and `y` axis, we return `true` to tell the game the missile is at its target. Otherwise, we return `false`. Add this before the `fire()` call:

```
function missile:move()
     local missileX, missileY = missile.x, missile.y
     missileX = missileX - deltaX
     missileY = missileY - deltaY
     missile.x, missile.y = missileX, missileY
     local diffX = math.abs(targetX - missileX)
     local diffY = math.abs(targetY - missileY)
     if diffX < 5 and diffY < 5 then
           deltaX, deltaY = 0, 0
           return true
     end
     return false
end
```

7. The last thing we'll add is the `destroy()` function. All we need to do in this case is remove the missile from the display list:

```
function missile:destroy()
     missile:removeSelf()
     missile = nil
end
```

Ready, Aim, Fire

Next, we'll add the listener that will handle touch events and tell the Missile module, where the missile should be fired. Touch event handlers in Corona listen for (oddly enough) "touch" events, so we need to create an event listener and callback function for that. Touch events have different phases, so we'll wait until the player has lifted their finger off the screen and use the last point of contact as the target destination of the missile.

8. Declare the event handler function, which we'll call `fireMissile`:

```
-- functions
local initGround, initScore
local configureLevel, startListeners, run, fireMissile
```

9. Add the event listener to the `startListeners()` function:

```
function startListeners()
     Runtime:addEventListener("enterFrame", run)
     Runtime:addEventListener("touch", fireMissile)
end
```

10. Create the `fireMissile()` function below the `moveObjects()` function. When the touch event has reached the “ended” phase, we’ll create a new missile and add it to the `missiles` display group. We’ll pass to the missile `targetX` and `targetY` values, which we’ll obtain from the event object. We’ll also pass in origin values, which is the point from which the missile should start moving. In this case, we’ll use the bottom centre of the screen. We’ll also decrement the score by 25 points:

```
function fireMissile(event)
     if event.phase == "ended" then
            local missile = Missile.new(event.x, event.y, ↩
HALF_WIDTH + STAGE_LEFT, STAGE_TOP + STAGE_HEIGHT)
            missiles:insert(missile)
            setScore(-25)
     end
end
```

We’ll create a new display group that will contain all of the fired missiles, and incorporate that into our `moveObjects()` function. Because missiles work slightly different from bombs, we need to take some extra steps when moving them. The missiles are going to explode when they reach their destination (and we’ve added that capability), so we need to know when that’s happened. We know this happens when the missile’s `move()` function returns `true`. When it happens, we’ll remove the missile and trigger an explosion animation at that location.

11. Declare a new display group to hold the missiles:

```
-- display groups
local bombs
local missiles
```

12. In the `startGame()` function, initialize the display group and add it to the game display group:

```
function game.startGame()
     initGround()
     bombs = display.newGroup()
     game:insert(bombs)
     missiles = display.newGroup()
     game:insert(missiles)
     configureLevel()
     startListeners()
end
```

13. In the `moveObjects()` function, we’ll iterate through the `missiles` display group and call the `move()` function that missiles will have. If they’ve reached their target, we’ll create an explosion. Add the code in bold to `moveObjects()`:

```
function moveObjects()
     for i = bombs.numChildren, 1, -1 do
            local bomb = bombs[i]
```

```
                bomb()
            end
            for i = missiles.numChildren, 1, -1 do
                local missile = missiles[i]
                local atTarget = missile:move()
                if atTarget == true then
                    missile:destroy()
                end
            end
        end
```

14. Lastly, we need to import the missile explosion module and the missile module that we wrote:

```
        -- modules
        local Bomb = require("Bomb")
        local Missile = require("Missile")
```

If you test out the game now, you should be able to fire missiles. You will get an error because of the attempt to call the so-far-nonexistent `setScore()` function, so we'll add that now.

Keeping Score

Players earn points by destroying bombs, and it will cost them points to fire missiles. The former may be obvious; the latter is intended to deter the player from wantonly firing missiles all over the screen, and we've already started adding that with the call to `setScore()` when the player fires a missile. Players will earn more points if they are able to destroy bombs as soon as possible after the bombs have been dropped (i.e. when they are higher up on the screen). This will reward faster reaction times.

15. Copy the "ArmorPiercing.ttf" font file into your project folder.

16. First, we'll declare all of the variables we need to implement a scoring system. These include a variable for the score, the display object showing the score, a function to initialize the score display object, and a function to actually change the score. Add the code shown in bold to the appropriate locations in Game.lua:

```
-- display objects
local sky, ground, groundPhysics, scoreText

-- variables
local frameCounter, bombSpeed, bombDropTimes
local bombsThisLevel = 5
local levelDuration = 600
local currentLevel = 0
local score = 0

-- functions
local initGround, initScore
local configureLevel, startListeners, run
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit
local setScore
```

17. Add the call to `initScore()` to the `startGame()` function:

```
function game.startGame()
     initGround()
     initScore()
     bombs = display.newGroup()
     game:insert(bombs)
     configureLevel()
     startListeners()
end
```

18. Add the `initScore()` function above the `initGround()` function. We'll use the `display.newText()` function to create the `scoreText` display object and centre it at the top of the screen. One legacy issue with the `newText()` function is that although we can specify the position of the text field in the constructor function (the two zeroes we're passing in), this position is relative to the top left corner of the text field. However, when we set the `x`/`y` properties of the text object later on, this will be relative to the middle of the object. Since the latter will make it easier to position the text field in the exact middle of the screen, we'll actually set the text object's position twice. We'll also use the custom font we included in our `build.settings` file way back in Exercise 2:

```
function initScore()
     scoreText = display.newText(game, "0", 0, 0, "ArmorPiercing", 48)
     scoreText.x = _G.HALF_WIDTH
     scoreText.y = 20
end
```

19. Now, we can create the `setScore()` function. This will take a single parameter, which is the amount by which the score should change. The function will be responsible for updating the score variable as well as the text object displaying the score. Add the function just after `groundHit()`:

```
function setScore(amount)
     score = score + amount
     scoreText.text = score
end
```

Collisions

Checking for collisions with missiles is simply a matter of extending `collisionHandler()` to check for different ids, and responding accordingly. Part of what we'll need to do is figure out how many points the player has scored by destroying the bomb.

20. Declare the new functions we'll need:

```
-- functions
local initGround, initScore
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit
local calculateScore, setScore
local shakeScreen
local gameOver
```

21. The score will be calculated based on the location of the bomb when it's destroyed, so we'll see a constant, `BASE_SCORE`, that we'll use in that calculation:

```
-- constants
local BASE_SCORE = 50
local BASE_DROP_TIME = 400
```

22. In the `collisionHandler()` function, expand the `if` statement to include missiles. If a missile is hit, we'll call the `calculateScore()` function and then call `missileHit()`. Because the number of points a player earns depends on the `y` position of the bomb when it is destroyed, we need to pass in the bomb's `y` property to that function. For the other function, we'll pass both the bomb and the object it hit:

```
function collisionHandler(self, event)
     if event.phase == "began" then
           local objectHit = event.other
           local id = objectHit.id
           if id == "ground" then
                 groundHit(self)
           elseif id == "missile" then
                 calculateScore(self.y)
                 missileHit(self, objectHit)
           end
           if #bombDropTimes == 0 then
                 configureLevel()
           end
     end
end
```

23. Add the `calculateScore()` function above `setScore()`. To calculate the actual score, we'll multiply the `BASE_SCORE` constant by some number between 1 and 2. The higher on the screen the bomb is when it's destroyed, the closer to 2 that multiplier should be:

```
function calculateScore(bombY)
     local heightMultiplier = 1 + (_G.STAGE_HEIGHT + ↩
_G.STAGE_TOP - bombY) / (_G.STAGE_HEIGHT + _G.STAGE_TOP)
     local scoreToAdd = math.floor(BASE_SCORE * heightMultiplier)
     setScore(scoreToAdd)
end
```

24. After the `groundHit()` function, add `missileHit()`. For now, we'll just remove both display objects:

```
function missileHit(bomb, missile)
     missile:destroy()
     bomb:destroy()
end
```

6. Creating the Sprite Sheet

In this exercise you will use Zwoptex to create the sprite sheets that will contain the explosion animations.

Zwoptex is available for OS X only. While there probably are similar sprite sheet creation programs available for Windows, I'm not personally familiar with them.

Zwoptex

In the interest of time, we'll skip the process of generating the frames for all of the sprites used in the game. Instead, we'll use the images in the 'sprite images' folder in the workshop files.

1. Start Zwoptex and create a new file. Save it as "SpriteSheet.zwd" in the 'sprite images' folder.
2. Click the "Publish Settings" button on the right side of the Zwoptex window.
3. Under "Texture", click the "Browse" button beside "Save to File", and set the target directory to the project folder. Make sure that the filename is "SpriteSheet.png".
4. Repeat step 3 for the "Coordinates", using a filename of "SpriteSheet.lua".
5. Click "Done".
6. Click the "Import" button in the Zwoptex toolbar and navigate to the "sprite images" folder.
7. Select all of the files in the folder by pressing Command-a on the keyboard, and then click "Open".
8. In the panel on the left side of Zwoptex, configure the sprite sheet size and layout:
    a. Set the width to 512 pixels and the height to 256 pixels.
    b. Select the "Max Rect" sort algorithm.
9. Click the "Layout" button in the toolbar.

10. Click the “Publish” button.

There should now be two new files in the project folder, “SpriteSheet.png” and “SpriteSheet.lua”. We’ll use this sprite sheet for lower-resolution devices like the iPhone 3. The .png file is the sprite image file, and the Lua file contains the sprite sheet data.

I’ve gone ahead and created a sprite sheet for the iPhone4, but if you’d like to try creating another sprite sheet, feel free to repeat this process with the images in the “sprite images hi-res” folder. Just make sure that you save the files as “SpriteSheet@2x.png” and “SpriteSheet@2x.lua”.

Setting Device-Specific Sheets

Unfortunately, the sprite API in Corona doesn’t easily handle dynamic sprite sheets. While we could simply use a sprite sheet sized for the iPhone 3 and let it scale up on higher-resolution devices, the result can look blurry.

To work around this, we’ll use two different sprite sheets, one for the iPhone 3, and one for higher-resolution devices - technically, it will be sized for the iPhone 4, but will work equally well on the iPad or Android devices and tablets.

The trick in using the high-res sprite sheet relies on the fact that while scaled-up graphics don’t look very good, *scaled-down* graphics look just great. To understand why we’ll need to scale the high-resolution sprite sheet down, consider what’s happening in the application when it’s run on an iPhone 4:

- The application is configured for a 320 x 480 screen
- When run on an iPhone 4, the Corona engine knows that it needs to scale the graphics up (doubling them in size) to fit the iPhone 4’s 640 x 960 screen
- Since we’re using a high-resolution sprite sheet that is already double the size required for a 320 x 480 screen, the images will look twice as big as they should on an iPhone 4.
- As a result, we need to scale the graphics down (to half of their size) so that they are the right size.

This may seem like a lot of work, but in actual practice it only amounts to a few extra lines of code whenever we work with sprite sheets.

7. Explosions

Now that bombs are landing and missiles are firing, we need to make them blow up. In this exercise, we'll create explosion animations from our sprite sheet and trigger them when a bomb hits the ground or when a missile.

The explosions will have slightly different requirements, but they're not so complex that they need to be broken into separate modules:

- The explosion associated with the bomb will occur when the bomb hits the ground (or, later on, a target on the screen), so we need to assure that it's aligned with the bottom of the screen
- The missile explosion will occur when a missile reaches its destination, or when it collides with an incoming bomb. Like the missile, the missile explosion will be able to destroy incoming bombs. So we need to turn that explosion into a physics object and incorporate it into our collision handler.

Which Sprite Sheet?

We have two sprite sheets, one for lower-resolution devices like the iPhone 3, and one for devices like the iPhone 4 and others with a higher resolution. Unfortunately, Corona does not have a simple way of telling it which sprite sheet to use, so we need to do a bit of work on that front.

1. If they're not there already, copy the sprite sheet image and data files into your project folder (copy the files from the supplied assets if required).

2. Create a new module and save it as "Explosion.lua". Since we know we'll be create a sprite sheet, we need to `require()` the sprite module, which is built into Corona.

    ```
    local Explosion = {}

    local sprite = require("sprite")

    return Explosion
    ```

3. Next, we need to determine which sprite sheet image and data module to use. To figure this out, we'll look at the value of `display.contentScaleX`. When it's equal to 1, we know that we should use our "regular" sprite sheet; when it's less than 1, we need to use our high-resolution sprite sheet:

```
local image, dataModule

if display.contentScaleX == 1 then
     image = "SpriteSheet.png"
     dataModule = "SpriteSheet"
elseif display.contentScaleX < 1 then
     image = "SpriteSheet@2x.png"
     dataModule = "SpriteSheet@2x"
end
```

4. Next, we can create the sprite sheet and add a couple of sprites, one for the bomb explosion and one for the missile explosion:

```
local zwoptexData = require(dataModule)
local data = zwoptexData.getSpriteSheetData()
local spriteSheet = sprite.newSpriteSheetFromData( image, data )
local spriteSet = sprite.newSpriteSet(spriteSheet, 1, 42)
sprite.add(spriteSet, "bomb", 1, 21, 1, 1)
sprite.add(spriteSet, "missile", 22, 21, 1, 1)
```

5. Now, we can create our `Explosion.new()` function. We'll pass in a few parameters: the type of explosion to create, and the `x` and `y` properties of the explosion. We'll also create our sprite display object from the sprite set we'd previously created:

```
function Explosion.new(explosionType, explosionX, explosionY)

     assert(explosionType or explosionX or explosionY, ↩
"Required parameter missing")

     local explosion = sprite.newSprite(spriteSet)
     explosion.id = "explosion"

     return explosion
end
```

6. To make sure that the sprites are scaled properly on high-resolution devices we'll scale the sprite down if required. Add this before the `return` statement:

```
if display.contentScaleX < 1 then
     explosion.xScale = 0.5
     explosion.yScale = 0.5
end
```

7. Next, we need to add some specific settings depending on the type of explosion being created. If it's a bomb explosion, we'll modify the reference point so that we can easily position the explosion along the bottom of the screen. If it's a missile explosion, we'll turn it into a physics object. We need to delay that step briefly, because adding a physics body to the world while a collision is being resolved can cause problems. To do this, we'll use Corona's timer library. We'll also add some variation by rotating the explosion some random amount:

```
if explosionType == "bomb" then
     explosion:setReferencePoint(display.BottomCenterReferencePoint)
else
     timer.performWithDelay(0, addPhysics)
     explosion.rotation = math.random(360)
end
```

A delay of 0 may imply that the function should be called right away, i.e. in the same frame as the performWithDelay() call. However, Corona will wait until the next frame before checking to see if any delayed function calls must be made. While we could set this to some non-zero number, setting it to 0 makes it clear (to me, at least!) that what we're telling Corona to do is, "call this function as soon as you can, *but not right this frame*." Setting it to something like 30 or 50 milliseconds may accomplish the same goal, but using a non-zero value implies that we have some specific point in mind, when really we don't care if the next frame is 20, 30, or 50 milliseconds from now.

8. We need to create the `addPhysics()` function required by the missile explosions. Add this after `spriteEventHandler()`. We'll use the `radius` property to create a circular physics object, which will roughly approximate the size of the explosion. Add this function before the `if` chunk:

```
local function addPhysics()
     physics.addBody(explosion, "static", {isSensor = true, radius =
15})
end
```

9. Now, we can return to some common settings. We'll position the sprite based on the parameters passed in, add an event listener, and the prepare and play the proper animation. Add this after the `if` chunk:

```
explosion.x = explosionX
explosion.y = explosionY

explosion.sprite = spriteEventHandler
explosion:addEventListener("sprite", explosion)
explosion:prepare(explosionType)
explosion:play()
```

10. We've added an event listener for the "sprite" event. The "end" phase of that event occurs when a sprite animation has finished playing. Add this function before the `if` statement:

```
local function spriteEventHandler(self, event)
     if event.phase == "end" then
           explosion:removeEventListener("sprite", explosion)
           explosion:removeSelf()
           explosion = nil
     end
end
```

Adding Explosions to the Game

Now we can incorporate the Explosion module into the game.

11. We need to `require()` the Explosion module in `Game.lua`:

```
-- modules
local Bomb = require("Bomb")
local Missile = require("Missile")
local Explosion = require("Explosion")
```

12. Since the missile explosions are physics objects, we'll add an `explosionHit` function:

```
-- functions
local initGround, initScore
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit, explosionHit
local calculateScore, setScore
```

13. A missile explosion will occur when a missile has reached its destination. We know this happens in the `moveObjects()` function when `atTarget` is `true`. Add this code in bold to `moveObjects()`:

```
function moveObjects()
     for i = bombs.numChildren, 1, -1 do
           local bomb = bombs[i]
           bomb:move()
     end
     for i = missiles.numChildren, 1, -1 do
           local missile = missiles[i]
           local atTarget = missile:move()
           if atTarget == true then
                 local explosion = Explosion.new("missile", ↩
missile.x, missile.y)
                 game:insert(explosion)
                 missile:destroy()
           end
     end
end
```

14. A missile explosion will also need to be triggered when a missile hits a bomb, so we need to add the code in bold to `missileHit()`:

```
function missileHit(bomb, missile)
     local explosion = Explosion.new("missile", missile.x, missile.y)
     game:insert(explosion)
     missile:destroy()
     bombs:remove(bomb)
end
```

15. The bomb explosion is only triggered when a bomb hits the ground, so can add a new explosion when `groundHit()` is called:

```
function groundHit(bomb)
     local explosion = Explosion.new("bomb", bomb.x, ↩
STAGE_HEIGHT + STAGE_TOP)
     game:insert(explosion)
     bomb:destroy()
end
```

16. Lastly, collisions between bombs and missile explosions need to be incorporated into our `collisionHandler` function. Add the code in bold:

```
function collisionHandler(self, event)
     if event.phase == "began" then
            local objectHit = event.other
            local id = objectHit.id
            if id == "ground" then
                  groundHit(self)
            elseif id == "missile" then
                  calculateScore(self.y)
                  missileHit(self, objectHit)
            elseif id == "explosion" then
                  calculateScore(self.y)
                  explosionHit(self, objectHit)
            end
            if #bombDropTimes == 0 then
                  configureLevel()
            end
     end
end
```

17. And then add the `explosionHit()` function after `missileHit()`. Since a bomb can't "destroy" an explosion, all we need to do is remove the bomb:

```
function explosionHit(bomb, explosion)
     bombs:remove(bomb)
end
```

8. Targets

Now that things are blowing up nicely, we need to give bombs something to hit and the player something to protect.

The targets will be a number of lunar landers sitting on the surface of the moon. The steps we need to implement to get the targets in place are as follows:

- Select one of the lander images to use, and place it correctly on the screen;
- Turn it into a physics body so that we can detect collisions between bombs and landers;
- Hone the bomb targeting algorithm - or really, create one, since we're just randomly targeting the bombs right now;
- Handle collisions between bombs and landers.

Lander Location

First, we'll get the landers positioned on the screen. We'll add the code that will determine the lander location to the Game module. The code for the landers will be in a separate module.

1. We'll create a separate initialization function for the landers called, appropriately enough, `initLanders()`. So, declare that along with the other functions in Game.lua:

```
        -- functions
        local initGround, initScore, initLanders
```

2. We'll also store the landers in their own display group, which will give us an easy method of keeping track of how many landers are left. When the number of landers reaches zero (i.e. in Lua, `landers.numChildren == 0`), then we know the game is over. Declare the `landers` display group:

```
    -- display groups
    local bombs
    local missiles
    local landers
```

3. We'll call the `initLanders()` function when a game is started, so add that function call to the `game:startGame()` function:

```
function game:startGame()
     initGround()
     initScore()
     initLanders()
     bombs = display.newGroup()
     game:insert(bombs)
     missiles = display.newGroup()
     game:insert(missiles)
     configureLevel()
     startListeners()
end
```

4. Now we can start writing the function itself. Below the `initGround()` function, create the function and initialize the `landers` display group:

```
function initLanders()
     landers = display.newGroup()
     game:insert(landers)
end
```

5. We'll place six landers on the moon, an equal distance from each other. First, we need to determine that distance between each. So that they're evenly spaced across the screen, we'll position the first lander at an `x` position of half that distance. For example, on an iPhone 3, the six landers would be 80 pixels from each other (480 pixels / 6 landers). So the first pixel would have an x position of 40. Overall, the landers would be positioned at 40, 120, 200, 280, 360, and 440, which will create the even spacing. Of course, the game may be played on other devices so we need to calculate this based on the actual available screen size. As part of this calculation, we'll offset `currentPosition` by the actual left side of the stage, which may vary depending on the device. Regardless of the screen size, after we've determined the position of the first lander, all we need to do is add the distance between landers to that position for every lander we create. The code shown in bold will do that for us:

```
     function initLanders()
           landers = display.newGroup()
           game:insert(landers)
           local distanceBetweenLanders = STAGE_WIDTH / 6
           local currentPosition = STAGE_LEFT + ↩
distanceBetweenLanders / 2
           for i = 1, 6 do
                 local lander = Lander.new(currentPosition)
                 landers:insert(lander)
                 currentPosition = currentPosition + ↩
distanceBetweenLanders
           end
     end
```

6. That takes care of the positioning. One thing that you may have noticed is that we're calling `Lander.new()`, even though we haven't created that module yet. So, if you try

to test the code at this point, you'll get an error. Let's `require()` the module now, even though we haven't created it, so that we can test more easily as we create the Lander module. Add the code in bold under the other modules we're importing:

```
-- modules
local Bomb = require("Bomb")
local ExplosionBomb = require("ExplosionBomb")
local Lander = require("Lander")
```

Landers

Now we're ready to start work on the Lander module.

7. Create a new file and save it as "Lander.lua". Add the requisite module code:

```
local Lander = {}

return Lander
```

8. Now we can start our `new()` function. We're passing in the `x` position of the lander, so we need to include that in our constructor function:

```
function Lander.new(landerX)

     assert(landerX, "Required parameter missing")

end
```

9. There are four different lander images to select from. We'll pick an image at random, and then use that create the lander image. We'll also position the lander using the value passed in, and place it at the bottom of the screen:

```
function Lander.new(landerX)

     assert(landerX, "Required parameter missing")

     local imageNumber = math.random(4)
     local imageFile = "lander" .. imageNumber .. ".png"

     local lander = display.newImageRect(imageFile, 50, 41)
     lander.id = "lander"

     local halfHeight = lander.height / 2
     local halfWidth = lander.width / 2

     lander.x = landerX
     lander.y = _G.STAGE_TOP + _G.STAGE_HEIGHT - halfHeight

     return lander

end
```

10. Next, we can create the physics shape for the lander. We'll make the shape a rectangle, five pixels high, positioned along the bottom edge of the lander. Here's what the shape table looks like - add this before the `return` statement:

```
local physicsShape = {
     -halfWidth, halfHeight - 5,
     halfWidth, halfHeight - 5,
     halfWidth, halfHeight,
     -halfWidth, halfHeight
}
```

11. Next, we can create the new physics body:

```
physics.addBody(lander, "static", {isSensor = true, shape = ↩
physicsShape})
```

12. As with other physics objects, we'll add a `destroy()` function. To make things a little more interesting, we'll add a small animation to the lander before it's removed altogether. After the transition is complete, we'll remove the lander altogether. Add the following code before the `return` statement:

```
function lander:destroy()
     local function onComplete(event)
          lander:removeSelf()
          lander = nil
     end
     transition.to(lander, {time = 500, y = STAGE_HEIGHT + ↩
lander.height, onComplete = onComplete})
end
```

Detecting collisions

With that in place, let's return to the game and add the landers to our collision detection handler. When a bomb hits a lander, we'll need to trigger the bomb's explosion animation and remove the lander from the game. Just as we handled collisions with other objects in their own functions, we'll do the same with lander collisions.

13. In `Game.lua`, find the list of function declarations, and add `landerHit`, shown in bold:

```
-- functions
local initGround, initScore, initLanders
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit, explosionHit, landerHit
local calculateScore, setScore
```

14. Find the collision handler function, and add a new condition to the `if` statement, checking for a lander hit. If a lander is hit, then we'll call `landerHit()`. Since we already have a function to show the bomb's explosion, we can call that, too:

```
function collisionHandler(self, event)
     if event.phase == "began" then
          local objectHit = event.other
          local id = objectHit.id
          if id == "ground" then
               groundHit(self)
          elseif id == "missile" then
               calculateScore(self.y)
               missileHit(self, objectHit)
          elseif id == "explosion" then
               calculateScore(self.y)
               explosionHit(self, objectHit)
          elseif id == "lander" then
               groundHit(self)
               landerHit(objectHit)
          end
          if #bombDropTimes == 0 then
               configureLevel()
          end
     end
end
```

15. Now we can add `landerHit()`, after the `explosionHit()` function previously created. We'll set this to call a new function in the Lander module, `lander:destroy`:

```
function landerHit(lander)
     lander:destroy()
end
```

Smart(er) Bombs, and Game Over

Things work pretty well now, but the bomb targeting is still completely random. Let's tweak that a little bit by giving the `dropBomb()` function a modicum of intelligence. Instead of picking a target `x` position completely at random, we'll first select a lander and then pick a random `x` position from a range of values on either side of the lander. For example, if a lander's `x` position is at 300, we can select a value between 250 and 350. This increases the likelihood of hitting a target, but doesn't guarantee it. While we're here, we'll also add a check to make sure there are any landers left before dropping a bomb, just to avoid any errors as a result of a bomb getting dropped at the same time the last lander is destroyed (an unlikely but possible situation).

16. We'll create a constant that will store the range of random numbers to select when picking a target. This will be the number of pixels on either side of the lander from which to select a target x position (so, the lower the number, the greater the accuracy).

```
-- constants
local BASE_SCORE = 50
local BASE_DROP_TIME = 400
local TARGET_RANGE = 50
```

17. Find the `dropBomb()` function and delete the line of code assigning a random value to `targetX`. First, we'll prevent further execution of the function if there are no targets remaining. Then, we'll select a lander at random from the landers display group, and use its `x` position to select a `targetX` value. Add the code shown in bold to the list of constants:

```
function dropBomb()
     if landers.numChildren == 0 then
          return
     end
     local index = math.random(landers.numChildren)
     local target = landers[index]
     local leftSide = target.x - TARGET_RANGE
     local rightSide = target.x + TARGET_RANGE
     local targetX = math.random(leftSide, rightSide)
     local bomb = Bomb.new(targetX)
     bombs:insert(bomb)
     bomb:fire(bombSpeed)
     bomb.collision = collisionHandler
     bomb:addEventListener("collision", bomb)
end
```

Now when you test the game out, the accuracy of the bombs should be somewhat improved. The last thing we'll add in this chapter is a simple “Game Over” screen when all the landers are destroyed.

18. From the “images” folder in the workshop files, copy the two “Game Over” png files into your project folder.

19. Declare a couple of new functions, `gameOver` and `stopListeners`, along with the other functions already declared:

```
-- functions
local initGround, initScore, initLanders
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit, explosionHit, landerHit
local calculateScore, setScore
local gameOver, stopListeners
```

20. In the `run()` function, add a check to see if there are any landers left. If there aren't, then we know that the game is over and we can call the `gameOver()` function:

```
function run()
      moveObjects()
      frameCounter = frameCounter + 1
      checkForBombDrop()
      if landers.numChildren == 0 then
            gameOver()
      end
end
```

21. After `startListeners()`, add the `stopListeners()` function. It just does the opposite of `startListeners()`:

```
function stopListeners()
     Runtime:removeEventListener("enterFrame", run)
     Runtime:removeEventListener("touch", fireMissile)
end
```

22. Above the `game:startGame()` function, add the `gameOver()` function. First, we'll call the stopListeners() function we just added:

```
function gameOver()
     stopListeners()
end
```

23. Next, we'll display the game over graphic, using the `display.newImageRect()` function to select a device-appropriate image, and display it in the centre of the screen:

```
function gameOver()
     Runtime:removeEventListener("enterFrame", run)
     local gameOverImage = display.newImageRect( game, ↩
"GameOver.png", 425, 79 )
     gameOverImage.x = HALF_WIDTH
     gameOverImage.y = HALF_HEIGHT
end
```

9. Sounds

In this exercise, we'll add some background music as well as a few sound effects. Adding sound effects is a straightforward process of preloading the sounds we want to play, and then playing them when required.

For the purpose of the workshop, we'll use .wav files. While the size of .wav files makes extensive use of them in an actual mobile project problematic, they have the benefit of cross-platform support in Corona's OpenAL audio engine. For more on the support of different audio formats in Corona, see http://developer.anscamobile.com/partner/audionotes

Background Music

We'll start by adding the background music, which we'll place inside the Game module.

1. Copy the sound effect files into your project folder.

2. At the top of the `new()` function, add a new section of declarations for our sound effects. We'll load sounds right away rather than wait until we actually need to play them, so we'll add the background music first. We'll stop the background music when the game is over, so we'll also declare a variable to use as an audio channel, which we'll need to stop the music:

```
-- sound effects, channels
local soundBackground = audio.loadStream("SoundBackground.mp3")
local soundBackgroundChannel
```

3. Next, start the music playing. We want the background music to keep looping while the game is being played; to do this, we pass a table as the second parameter, with the 'loops' property set to -1. In the `startGame()` function, add the code marked in bold:

```
function game:startGame()
     initGround()
     initScore()
     initLanders()
     bombs = display.newGroup()
     game:insert(bombs)
     missiles = display.newGroup()
     game:insert(missiles)
     configureLevel()
     startListeners()
     backgroundChannel = audio.play(soundBackground, {loops = -1})
end
```

The process for adding other sound effects requires the same basic steps. Some of these sounds will be added to the game module, others will be added to other modules as required.

4. In `Missile.lua`, add at the top of the module the sounds to be played when a missile is launched and when it explodes:

```
local MISSILE_SHAPE = {-6,-1, 8,-1, 9,-1, 8,1, -6,1, -11,4, -11,-5}
local BASE_SPEED = 4
local RADIANS_TO_DEGREES = math.pi / 180

local soundFire = audio.loadSound("SoundLauncher.wav")
local soundDestroy = audio.loadSound("SoundMissileDestroy.wav")
```

5. Inside the `fire()` function, play the appropriate sound effect:

```
local function fire()
     missile.x, missile.y = originX, originY
     local diffX = originX - targetX
     local diffY = originY - targetY
     missile.rotation = math.atan2(diffY, diffX) / ↩
RADIANS_TO_DEGREES + 180
     local distance = math.sqrt(diffX * diffX + diffY * diffY)
     local theta = math.asin(diffX / distance)
     deltaX = BASE_SPEED * math.sin(theta)
     deltaY = BASE_SPEED * math.cos(theta)
     audio.play(soundFire)
end
```

6. Do the same in the `destroy()` function:

```
function missile:destroy()
     audio.play(soundDestroy)
     missile:removeSelf()
     missile = nil
end
```

7. In `Bomb.lua`, we'll add a couple of sound effects: one for when a bomb is dropped, and one that will play when a bomb is destroyed. First, preload the sounds as we did with the Missile module:

```
local RADIANS_TO_DEGREES = 180 / math.pi
local BOMB_SHAPE = {-6,-1, 8,-1, 9,-1, 8,1, -6,1, -11,4, -11,-5}

local soundBombDrop = audio.loadSound("SoundBombDrop.wav")
local soundBombDestroyed = audio.loadSound("SoundBombDestroyed.wav")
```

8. Play `soundBombDrop` when the `fire()` function is called:

```
function bomb:fire(dropTime)
     bombX = math.random(_G.STAGE_WIDTH)
     bombY = -100
     targetY = _G.STAGE_HEIGHT
     local diffX = targetX - bombX
     local diffY = targetY - bombY
     deltaY = diffY / dropTime
     deltaX = diffX / dropTime
     local targetAngle = math.atan2(diffY, diffX) * RADIANS_TO_DEGREES
     bomb.rotation = targetAngle
     audio.play(soundBombDrop)
end
```

9. Then, play `soundBombDestroyed` when the `destroy()` function is called:

```
function bomb:destroy()
     audio.play(soundBombDestroyed)
     bomb:removeEventListener("collision", bomb)
     bomb:removeSelf()
     bomb = nil
end
```

10. The next sound effect to add will play when a lander is destroyed. In the Lander module, load the required sound:

```
local Lander = {}

local soundLanderDestroyed = audio.loadSound("SoundTargetHit.wav")

function Lander.new(landerX)
```

11. And play it:

```
function lander:destroy()
     local par = lander.parent
     local function onComplete(event)
           lander:removeSelf()
           lander = nil
     end
     transition.to(lander, {time = 500, y = STAGE_HEIGHT + ↩
lander.height, onComplete = onComplete})
     audio.play(soundLanderDestroyed)
end
```

12. The final sound we'll add will be a "game over" sound effect. Back in `Game.lua`, navigate to the `gameOver()` function, and add the code necessary to load and play

the sound. In this case, we won't preload the sound since it only plays at the very end of the game, when performance isn't really much of an issue. We'll also fade out the background audio track:

```
function gameOver()
     local gameOverSound = audio.loadSound("SoundGameOver.wav")
     audio.play(gameOverSound)
     audio.fadeOut{channel = soundBackgroundChannel, time = 2000}
     stopListeners()
     local gameOverImage = display.newImageRect( game, ↩
"GameOver.png", 425, 79 )
     gameOverImage.x = HALF_WIDTH
     gameOverImage.y = HALF_HEIGHT
end
```

Now, when you test the game, you should hear all of the different sounds.

A. Volume Control

We'll add a volume control slider that will be accessible via a button in the top corner of the screen. The button will toggle the visibility of the slider, which can be used to set the volume.

Both the slider and the toggle button to show/hide it will take advantage of the ability of Corona objects to dispatch custom events. Both components will dispatch an event when the user interacts with them; in the case of the toggle button, the event will signal the need to show or hide the slider. When the player moves the slider, it will dispatch an event with the new value of the slider. The game module, which will be listening for events from the components, will be configured to respond as required.

Creating the Slider Module

We'll code the slider first, then the toggle button, and then add both into the game.

1. Create a new file and save it as `Slider.lua`. Add the skeleton module code. We're going to provide the option to pass in a single parameter to the constructor which will let us set an initial value for the slider:

   ```
   local Slider = {}

   function Slider.new(initialValue)

   end

   return Slider
   ```

2. The `new()` function will require a total of three display objects. First, there is the display group that will contain the slider graphics. The slider itself will consist of two vector objects: the "track" and the "thumb". Declare all of these at the top of `new()`:

   ```
       local slider = display.newGroup()
       local track, thumb
   ```

3. We'll also need to declare three local functions. One will create the user interface, there will be a callback function for touch events (when the user manipulates the slider, and lastly we'll put the code that actually calculates the new slider value and dispatches an event signaling a change in the value into its own function.

   ```
       local createUI, sliderTouch, dispatch
   ```

4. Our last declaration will be a variable to store the value of half the width of the track:

```
        local halfTrack
```

5. Next, the `createUI()` function will actually create and position the graphics. The track and thumb will be simple vector objects. If there's an initial value, we'll position the thumb as required. To determine the location of the thumb, we multiply the initial value (which we assume to be between 0 and 1) by the width of the track. Because the default registration point in Corona is the middle of a display object, we subtract the value of `halfTrack` to obtain the necessary offset. Lastly, we'll add the "touch" event listener to the track.

```
    function createUI()

        track = display.newRoundedRect( slider, -100, -10, 200, 20, 10 )
        track:setFillColor(64, 64, 64)
        halfTrack = track.width / 2

        thumb = display.newCircle( 0, 0, 10 )
        thumb:setFillColor(0, 0, 128)
        slider:insert(thumb)

        if initialValue then
            thumb.x = track.width * initialValue - halfTrack
        end

        track:addEventListener("touch", sliderTouch)
    end
```

6. We can code the `sliderTouch` callback function next. During the "began" and "ended" phases we need to set and remove, respectively, focus on the track. This prevents the track from "losing" a touch event if the user moves their finger off the track during the touch. Next, we need to position the thumb to correspond to the user's touch. To do this we need to use the `contentToLocal()` function to convert the event's x/y coordinates, which are global, to coordinates local to the slider. Then, we can position the thumb properly along the track.

```
    function sliderTouch(event)
        if event.phase == "began" then
            display.getCurrentStage():setFocus(track)
        elseif event.phase == "ended" then
            display.getCurrentStage():setFocus(nil)
        end
        thumb.x = slider:contentToLocal(event.x, event.y)
        if thumb.x < track.x - halfTrack then
            thumb.x = track.x - halfTrack
        elseif thumb.x > track.x + halfTrack then
            thumb.x = track.x + halfTrack
        end
        dispatch()
    end
```

7. Next, we can calculate the new value for the slider and dispatch an event to notify any listeners that the slider's value has changed. Since the Corona audio API requires a value from 0 to 1 for the volume, we'll calculate the value within that range. Then, when we add the slider to the game, we can listen for a change to the slider value and simply set the audio volume to whatever value is passed in the event object.

```
function dispatch()
     local val = (thumb.x + track.width / 2)  / track.width
     slider:dispatchEvent{ name = "change", value = val}
end
```

8. The last thing we need to add is a function to destroy the slider if we no longer need it - in this particular case, we won't be using it, but it's nice to include this right away, which makes it easier to repurpose this in other projects where it may be required. We also need to call the `createUI()` function, and return the slider display group. This should appear immediately before the 'end' statement that closes the `new()` function:

```
function slider:destroy()
      track:removeEventListener("touch", sliderTouch)
      slider:removeSelf()
end

createUI()

return slider
```

That will serve our purpose nicely, and also demonstrates an easy way to create custom components that can be wired into other applications. At the moment, this slider is overly specific to our requirements in this game, but the module could easily be extended to make it more broadly useful in other contexts.

A Toggle Button to Access the Menu

Before we add the slider to the game, we'll create another module for a simple toggle button that will control the slider's visibility. The toggle button will have two different images (an "on" and an "off" button) that will alternate visibility as the user presses it. There will also be a "dispatch" function that will serve a purpose similar to that in the Slider module.

9. Copy the “MenuOn” and “MenuOff” images into your project folder.

10. Create a new file, save it as `MenuToggle.lua`, and add the standard module code:

```
local MenuToggle = {}
function MenuToggle()
end
return MenuToggle
```

11. We’ll declare all of the display objects and functions required; the functions required will be similar to those used in the Slider:

```
local toggle = display.newGroup()
local menuOnButton, menuOffButton
local createUI, dispatch, onTouch
```

12. Add the `createUI()` function. This will add the images to the display group. Since the “menu” will not be immediately visible, we’ll make sure that only the “on” button can be seen. We’ll also add the “touch” event listener for the button. Since the button is really a set of two buttons whose visibility will be toggled on and off, we’ll add the listener to the entire display group; that way, any object that is pressed will result in the touch event getting dispatched:

```
function createUI()
     menuOnButton = display.newImageRect( toggle, "menuOn.png", ↩
30, 30 )
     menuOffButton = display.newImageRect( toggle, "menuOff.png", ↩
30, 30 )
     menuOffButton.isVisible = false
     toggle:addEventListener("touch", onTouch)
end
```

13. Add the `onTouch()` event handler function. In the “ended” phase of the touch event, we’ll call the `dispatch()` function and then swap the visibility of the two button images, which will give the ‘toggle’ effect. This is using the ability of multiple assignment in Lua to easily transpose two values. Since one button’s `isVisible` property will always be `true` and the other’s will always be `false`, we can just swap those values between the two buttons:

```
function onTouch(event)
     if event.phase == "ended" then
            dispatch()
            menuOffButton.isVisible, menuOnButton.isVisible = ↩
menuOnButton.isVisible, menuOffButton.isVisible
     end
     return true
end
```

14. In `dispatch()`, we’ll use the visibility of `menuOnButton` to determine whether the menu should be shown or not. If `menuOnButton` is visible when the toggle button is pressed,

that means we want to show the menu; if it's not visible, then we want to hide it the menu:

```
function dispatch()
     local showMenu = menuOnButton.isVisible
     toggle:dispatchEvent{ name = "toggle", showMenu = showMenu}
end
```

15. Lastly, add a `destroy()` function, call the `createUI()` function, and return the toggle display group. This should all appear before the `end` statement:

```
function toggle:destroy()
     toggle:removeEventListener("touch", onTouch)
     toggle:removeSelf()
end

createUI()
return toggle
```

Adding the Components to the Game

To implement the volume control, we'll need to add both components to the game. When we do that, we'll also need to add event listeners for the events that will be dispatched by both components.

16. Before we can add the components, we need to require the modules we just created:

```
-- modules
local Bomb = require("Bomb")
local Missile = require("Missile")
local Explosion = require("Explosion")
local Lander = require("Lander")
local MenuToggle = require("MenuToggle")
local Slider = require("Slider")
```

17. Since we're going to be creating a couple of display objects, those should be declared as well:

```
-- display objects
local sky, ground, groundPhysics, scoreText
local menuToggle, slider
```

18. We'll also create a separate function that will create the menu toggle button, so we'll need to declare that as well. We also need callback functions for both components' events:

```
-- functions
```

```
local initGround, initScore, initLanders, initMenuToggle
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit, explosionHit, landerHit
local calculateScore, setScore
local gameOver, removeListeners
local onMenuToggle, onSliderChange
```

19. The `initMenuToggle()` function can go after the `initLanders()` function. First, we create the display object and insert it into the game display group. Then, we position it in the top right corner of the screen. The last thing we need to do is add an event listener to the display object, so that we can detect when it has dispatched its "toggle" event:

```
function initMenuToggle()
     menuToggle = MenuToggle.new()
     game:insert(menuToggle)
     menuToggle.x = STAGE_WIDTH - menuToggle.width
     menuToggle.y = menuToggle.height
     menuToggle:addEventListener("toggle", onMenuToggle)
end
```

20. Next, call the `initMenuToggle()` function when a new game is started. The order in which it is called will dictate its "layer" in the display list. The bombs and missiles will appear above the toggle button; if you're prefer that they didn't, you can call the function after the bombs and missiles display groups have been created:

```
function game:startGame()
     initGround()
     initScore()
     initLanders()
     initMenuToggle()
     bombs = display.newGroup()
     game:insert(bombs)
     missiles = display.newGroup()
     game:insert(missiles)
     configureLevel()
     startListeners()
     soundBackgroundChannel = audio.play(soundBackground, {loops =
-1})
end
```

21. We can now add the `onMenuToggle` function, which will be called when the "toggle" event is dispatched by the toggle button. The code will first check to see if the slider already exists. If it does not, it will be created, and we will pass in the current volume setting to the slider so that the thumb can be properly positioned. Next, the slider is added to the game display group and centered on the screen. We also need to add an event listener for the "change" event, so that we know when the slider value has changed from user interaction.

    The next part of the function will show or hide the slider as required. It will also call the `startListeners()` or `stopListeners()` function as required, which will have the effect of pausing or resuming the game:

```
function onMenuToggle(event)
     if not slider then
           slider = Slider.new(audio.getVolume())
           game:insert(slider)
           slider.x = HALF_WIDTH
           slider.y = HALF_HEIGHT
           slider:addEventListener("change", onSliderChange)
     end
     if event.showMenu == true then
           slider.isVisible = true
           stopListeners()
     else
           slider.isVisible = false
           startListeners()
     end
end
```

22. Lastly, we can add the `onSliderChange` function. This is the callback function for the “change” event listener we created in the previous step. All it needs to do is get the new value of the slider and pass that value to the audio library’s `setVolume()` function:

```
function onSliderChange(event)
     audio.setVolume(event.value)
end
```

B. Screen Shaking

To make the explosions a little more earth-shaking (or more to the point, moon-shaking), let's add in some screen shaking when the bomb lands.

The screen shaking effect will be created by offsetting the entire game display group from its original position by a small random amount. By doing this over the course of a number of frames, it will create the shaking effect. This effect will play over the course of 30 frames, so the effect should last for about a second or so.

1. Declare a variable at the top of the Game module called `shakeCounter`. This will keep track of the number of frames remaining in the shake effect. We'll initialize it to 0 since we don't want the screen to be shaking when the game first starts:

```
-- variables
local frameCounter, bombSpeed, bombDropTimes
local bombsThisLevel = 5
local levelDuration = 600
local currentLevel = 0
local score = 0
local shakeCounter = 0
```

2. Next, declare a variable called `shakeScreen`, which will be the function that will actually do the shaking:

```
-- functions
local initGround, initScore, initLanders, initMenuToggle
local configureLevel, startListeners, run, fireMissile
local checkForBombDrop, dropBomb, moveObjects
local collisionHandler
local groundHit, missileHit, explosionHit, landerHit
local calculateScore, setScore
local gameOver, removeListeners
local onMenuToggle, onSliderChange
local shakeScreen
```

3. In the `run` loop, add a check to see if `shakeScreen()` needs to be called. It should be called whenever the value of `shakeCounter` is greater than 0:

```
function run()
     moveObjects()
     frameCounter = frameCounter + 1
     checkForBombDrop()
     if shakeCounter > 0 then
          shakeScreen()
     end
     if landers.numChildren == 0 then
          gameOver()
     end
end
```

4. Add the `shakeScreen` function after `moveObjects()`:

```
function shakeScreen()
end
```

5. Inside `shakeScreen`, we'll select random values for the game display group's `x/y` properties, and then apply those properties to the group. We'll select a number that could be either positive or negative, so that the screen can shake in all directions. We'll also decrement `shakeCounter` - otherwise the shaking will continue indefinitely after a bomb hits:

```
function shakeScreen()
     local randX = math.random(-3, 3)
     local randY = math.random(-3, 3)
     game.x = randX
     game.y = randY
     shakeCounter = shakeCounter - 1
end
```

6. Also inside `shakeScreen`, we need to reset game to its original position after the shaking has stopped:

```
function shakeScreen()
     local randX = math.random(-3, 3)
     local randY = math.random(-3, 3)
     game.x = randX
     game.y = randY
     shakeCounter = shakeCounter - 1
     if shakeCounter == 0 then
          game.x = 0
          game.y = 0
     end
end
```

7. The last thing we need to do is set the `shakeCounter` to a non-zero value when a bomb hits the ground. We can do this inside the `groundHit()` function:

```
function groundHit(bomb)
     local explosion = Explosion.new("bomb", bomb.x, STAGE_HEIGHT +
STAGE_TOP)
     game:insert(explosion)
     bomb:destroy()
     shakeCounter = 30
end
```

Now, when you test the game and the bombs land, you should see the shaking effect. You can increase the range of random numbers or the value of `shakeCounter` to modify the effect.