Extracted from:

Create Mobile Games with Corona

Build with Lua on iOS and Android

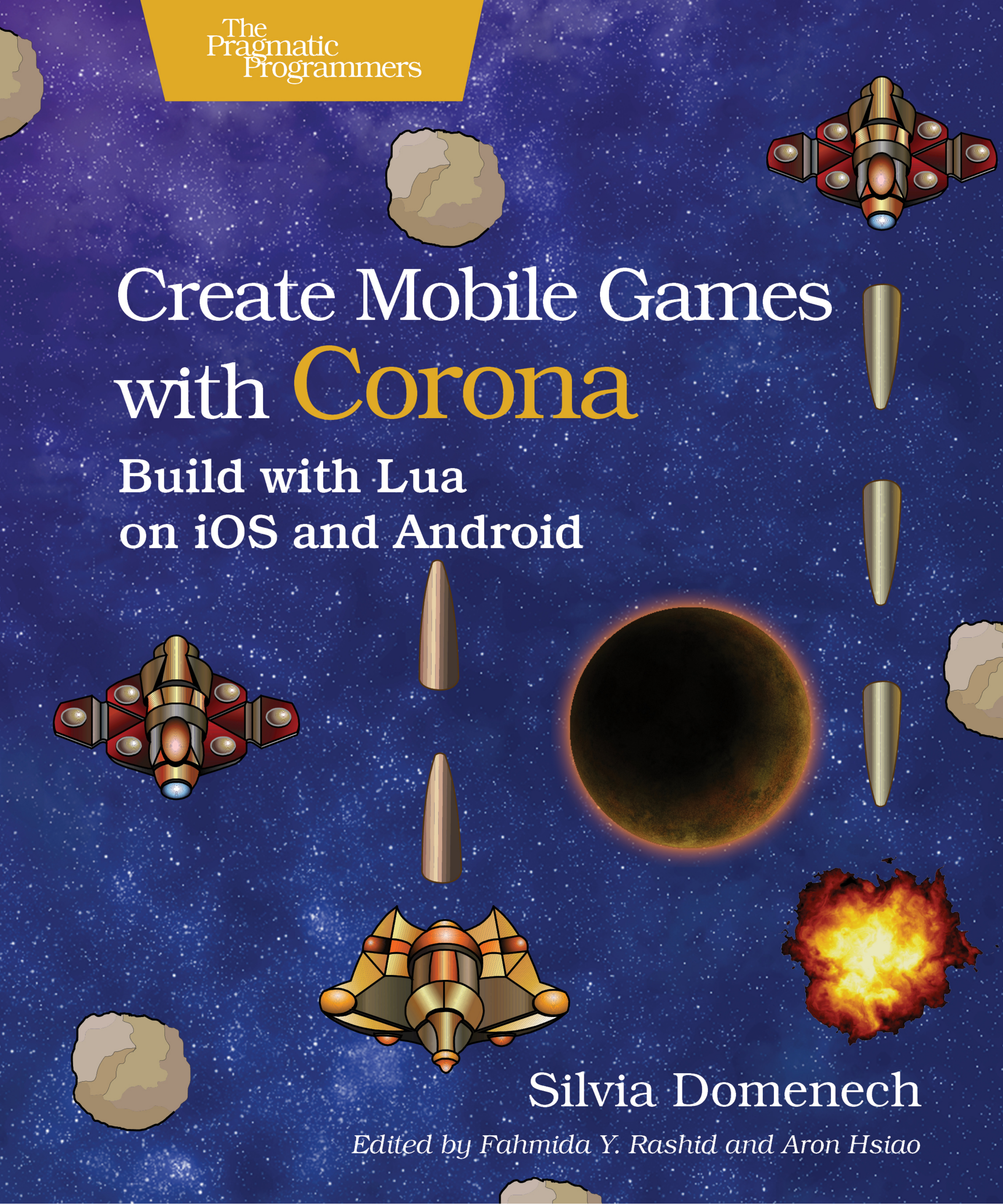
The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Create Mobile Games with Corona

Build with Lua on iOS and Android



Silvia Domenech

Edited by Fahmida Y. Rashid and Aron Hsiao

Create Mobile Games with Corona

Build with Lua on iOS and Android

Silvia Domenech

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Fahmida Y. Rashid and Aron Hsiao (editors)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-57-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P2.0—January 2014

2.3 Designing Our First Game: Planet Defender

Now that we've installed Corona and learned the basic programming vocabulary and techniques needed to develop games, it's time to start making a real game.

For our first project, we'll design a game that we can code and understand using the basic knowledge that we already have. We'll need to create and manage three main things.

- A way for the game to keep updating itself, frame after frame, in an endless loop that handles most of the game-related code (a game loop)
- A set of images to graphically represent the elements of the game (sprites)
- A few buttons and a start menu to enable the player to interact with the game (interactivity)

To make things easier on ourselves as beginners, we'll design a game with as little movement as possible—a space-themed game with enemy ships that fly in straight lines toward a planet at the bottom of the screen. The player will be in charge of defending the planet by destroying (tapping) the approaching ships. The game will end whenever an enemy ship reaches the planet.

Target Features

When designing a game, the best way to begin is by outlining a list of the features the game will have. For our first game, we'll need the following:

- A background image
- A properly positioned planet image
- Enemy ships that move toward the planet
- Functions to interpret and act on player input (screen taps) so that ships can be destroyed
- A score counter

2.4 Creating the Project

It's time to create a new project to hold our first game. Create a new multiscreen application. Rename the automatically generated scenetemplate.lua scene file to game.lua—we'll work only on the project's gameplay, so calling the code file game.lua makes it easier to recognize. It's important to change the scene names to something explicit because it makes coding much easier in the long run. Using scenetemplate.lua can be more comfortable now, but in five months' time we might not remember what the scene had, and we'll have to read it to know whether that's the file we want to update.

We also have to open main.lua to update the file we want to load first, because otherwise we'll get an error once the program attempts to open scenetemplate.lua. To update this, make sure that main.lua calls the scene name by using storyboard.gotoScene().

GameLoop/GameLoop/main.lua

```lua
-- Require the Storyboard API
local storyboard = require "storyboard"

-- load the game.lua scene
storyboard.gotoScene( "game" )
```

Now that the program will go directly to the game scene, it's time for us to start working on it.

Drawing the Background Image

Since it's a bit daunting to start using a new programming language without seeing results, let's begin by adding a background image to the game. As game developers, seeing images in our games is great because they let us see that the program isn't broken (or at least not completely). We'll sometimes have to make further checks to see whether the nonvisual code is working, but seeing images behave in the way we expect them to is usually a good sign.

We can draw an image on the stage by calling the display.newImage() function. Pass as a parameter the name of an image located in the game project folder, and Corona will load and display the image. To load an image called space_background.jpg, write display.newImage("space_background.jpg"). The function returns the image Corona has just loaded, so you'll usually want to store it as a local variable to be able to change its properties easily. You can save images in a folder to make your project folder neater, and then pass the relative path (for example, foldername/space_background.jpg) to this function.

When working with images, we can't just add them anywhere and hope they look good. In games, we usually work with many images, so we have to make sure each image is well-positioned. Also, when changing scenes, it's common to remove everything from the stage. To make it easier for us to access all objects on the stage, Corona has something called display *groups*. A group (or display group) in Corona is just like a folder on your PC—it helps you keep your images organized. Each scene automatically generated by Corona comes with a group variable called group where we can add our scene objects. We can use that whenever we work with images.

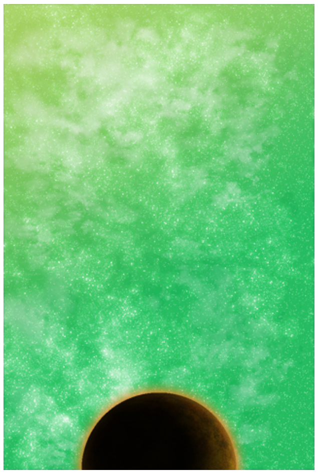


Figure 12—The game's background image

To add a display object to a group, call the group:insert() function, and the image will be added to the scene's main group. Now load a background image using display.newImage() and add it to the main scene group using group:insert().

You can set the image's anchor point, which is the point responsible for moving the image around, by setting the image's anchorX and anchorY variables. You can set values between 0 (top or left) and 1 (bottom or right). In this case, since we want to place the image's top left corner in the screen's top left corner, we have to set both values to 0.

GameLoop/GameLoop/game.lua

```lua
-- Load a background image when the scene is created
function scene:createScene( event )
    local group = self.view

    -- Load an image and add it to the scene's main group
    local image = display.newImage( "images/space_background.jpg" )
    image.anchorX, image.anchorY = 0, 0
    group:insert( image )
end
```

If you compile the project now, you'll be able to see the image in action, as shown in Figure 12, *The game's background image*.

Right now, it doesn't move or do anything, but it's always nice to see that the previously empty stage is no longer empty.

2.5 Coding the Game Loop

Having images on the screen is great, but we need a way to regularly update them to create the illusions of motion and activity. Otherwise, players will be bored by bullets, enemies, and a player character that just sit there, motionless. In this game, we have to move enemy ships and get rid of dead units (the ones that have been tapped).

Adding an Event Listener

Corona divides every second into a fixed number of up to sixty *frames*, which is the number of times that the stage is rendered and shown to the user. We can ask the program to call a function of our choosing whenever it's time for a new frame to appear. We tell Corona to do this using an *event listener* called enterFrame.

An event listener is like a little spy in Corona programs that will call a function for us (whichever function we ask it to call) whenever a specific event happens. In the case of the enterFrame listener, the event is the start of a new frame. Corona has other listeners that can call a function each time an animation changes, when a sound ends, or even when the player taps the screen.

We ask an event listener to do these things for us by calling the addEventListener() function. If we want an event listener to focus on just one object, we'll call object:addEventListener(), but when we want to track a generic event such as when the stage moves to the next frame, we'll use Runtime:addEventListener(), which assigns the listener to the game's running environment in general.

Many game developers call their frame-update functions either tick() or enterFrame(). We'll use tick() because it's easier to write and reminds us that time is passing with each update. We'll use the tick() name for all of our frame-update functions throughout the book, but remember that you can name the function in any way you like and it will still be called as long as you pass its name to the addEventListener() function along with "enterFrame".

Now write an empty tick() function to later call it through an enterFrame event listener.

GameLoop/GameLoop/game.lua
```
-- A placeholder for the tick function
--  Called every frame
function tick( )
    -- Here we'll add the code that needs to be executed each frame
end
```

Now that we know the type of event we'll add (enterFrame), the function we need to call to set up the event listener (Runtime:addEventListener()), and the function we want the event listener to call (tick()), add the event listener to the enterScene() function to call tick() sixty times per second. Remember that Corona generated enterScene() automatically when we created a new storyboard project and that the function is called as soon as the program enters the scene.

GameLoop/GameLoop/game.lua
```
-- Called immediately after scene has moved onscreen:
function scene:enterScene( event )
    local group = self.view
    -- Add an event listener
    -- This will call the tick function each frame:
    Runtime:addEventListener( "enterFrame", tick )
end
```

This enterFrame event listener is like a little CIA agent who will watch the game and call the tick() function, which is like the headquarters, every time the program enters a new frame. All we have to do now is add code to the tick() function to update all of the game's objects, doing whatever needs to be done in each frame to make this program behave like a game.

Adding EnterFrame Listeners to Objects

Corona lets us add event listeners to actual game objects. This means we can keep track of when the player touches an object in the game and if two objects collide with each other. We can also add an enterFrame event listener to game objects such as sprites, which you'll learn about in Chapter 3, *Sprites and Movement*, on page ?. However, if we add event listeners to objects that can be removed from the game, we need to make sure we also call the removeEventListener() function to remove the listeners. Otherwise, we might not clear it out properly, and it might be called for as long as the app remains active.

To avoid this, it's a good idea to avoid enterFrame events, consolidate actions in the game tick(), and make sure that we call relevant update functions for each of the objects on the stage. That way, the only time we'll have to remove an enterFrame event listener is if we ever leave the game scene. This also saves Corona from keeping track of lots of event listeners; if we're routinely tracking a series of recurrent events for lots of objects, then we can also consolidate them into a single event and loop through affected objects.

Updating the Game

Now we have a game that calls the tick() function regularly, but it doesn't actually do anything that a player can see yet because we haven't provided any instructions to Corona in the tick() function. Generally speaking, each time through tick() we'll check to see whether we need to add new objects (like enemy ships or bullets) to the game and whether a player has tapped an existing object, and we'll need to update all of the existing objects (move enemy ships, remove a ship that has been tapped, and so on).

The easiest way to structure a game loop is to make a mental list of all the objects in the game and then add a function call to them in the tick() function that updates each of them appropriately (whatever that may mean) in each frame. Planet Defender is a simple game, so the only objects that we need to update are the enemy ships. We'll also have to occasionally add new ships to the game, which adds a little wrinkle, but there's no "player ship" object in this game, or any other complications for the time being.

Structuring the Game Loop

Now that we've decided the tick() function will need to create enemy ships and update them as time passes, we can write a basic set of instructions using placeholder function calls for each of the actions we plan to add. This means we'll call the functions we need to have (as though we'd already written them), and then we can actually create each of the needed functions afterward. This makes it easy to have a manageable big-picture view of what we're doing and also keeps our tick() function clean and simple, even in complex games with lots and lots of updates and instructions. A game's tick() function is like its brain, so it's a good idea to keep it neat and organized so that it's easy to maintain and update the game as needed.

Let's start by adding placeholder calls to the tick() function. Add calls to functions named updateEnemies() and addEnemies(). updateEnemies() will update the enemy ships, and addEnemies() will add new enemies to the game.

GameLoop/GameLoop/game.lua
```lua
-- The tick function that will get called each frame
function tick( )
    -- Call several functions to update our game
    updateEnemies( )
    addEnemies( )
end
```

At this point, the game will try to call both of those functions each frame. Since we haven't written these yet, if we try to run the game, Corona will complain because it can't find the functions. Let's begin to fix this by writing an updateEnemies() function just before the createScene() function. This function needs to update the enemy ship positions on the screen to create the illusion of movement as time passes and to remove any ships that need to be removed. Since we haven't yet covered how to add ships, write a short comment describing what we want the function to do, and we'll get around to the actual code in in the next chapter. Also, print a short Enemies updated message on the console (this message will be printed once each frame) to see that the tick function is really being called.

GameLoop/GameLoop/game.lua

```
-- This function will update our enemies each frame
function updateEnemies( )
    print( "Enemies updated" )
end
```

Now write the addEnemies() function. We'll also put off the real work that goes into this function for now, so print another message saying Enemies added each time this function gets called.

Once you add this, running the code in the simulator should result in alternating messages on the console saying that the enemies have been updated and added, churning out very quickly. If these messages don't appear, it means that something (calls to a function, an event listener, and so on) is probably missing or mistyped. This simple debugging technique illustrates why the print() function is so useful.

You can also use a debugger from some of the integrated development environments Appendix 1, *Corona Resources*, on page ?.

GameLoop/GameLoop/game.lua

```
-- This function is called each frame and will add enemies to the game
function addEnemies( )
    print( "Enemies added" )
end
```

We've written the most basic game loop we can think of, yet it's a good way to see how to update games. This code is ready to be turned into something playable with the addition of a bit more code. We can add anything we want to the tick() function, or to any functions that it calls, and all of the instructions in them will be carried out each time Corona enters a new frame.

Configuring the Frame Rate

The game loop function is ready to do our bidding with each tick of Corona's clock, but we might want more or fewer ticks every second depending on whether we're coding fast-paced action or very slow-moving games. We can set the number of frames per second (FPS) that will occur in the game by updating the program's build.settings configuration file and assigning a value to the fps variable. When we set this variable to either 30 or 60 (the two choices that Corona accepts), the app will enter a new frame either thirty or sixty times per second, respectively.

GameLoop/GameLoop/build.settings

```
-- Change our game's frame rate to 30 or 60 (30 in this case)
fps = 30,
```

For this game, we'll keep the value at 30, which is the default value if we don't change anything. In any case, it's a good idea to test both speeds to see the difference in a game.

2.6 What We Covered

In this chapter, we covered how to create a game loop and event listeners, which are the fundamental frameworks that we'll use to build all of the games in this book. We saw how a series of functions called from the game loop come together into something that players experience as a game. We created a game project for our first game, Planet Defender, and wrote placeholder functions for enemy-ship creation and updates. In the next chapter, we'll build on this chapter's work, adding spaceships and other things that will begin to turn our project into an entertaining game.