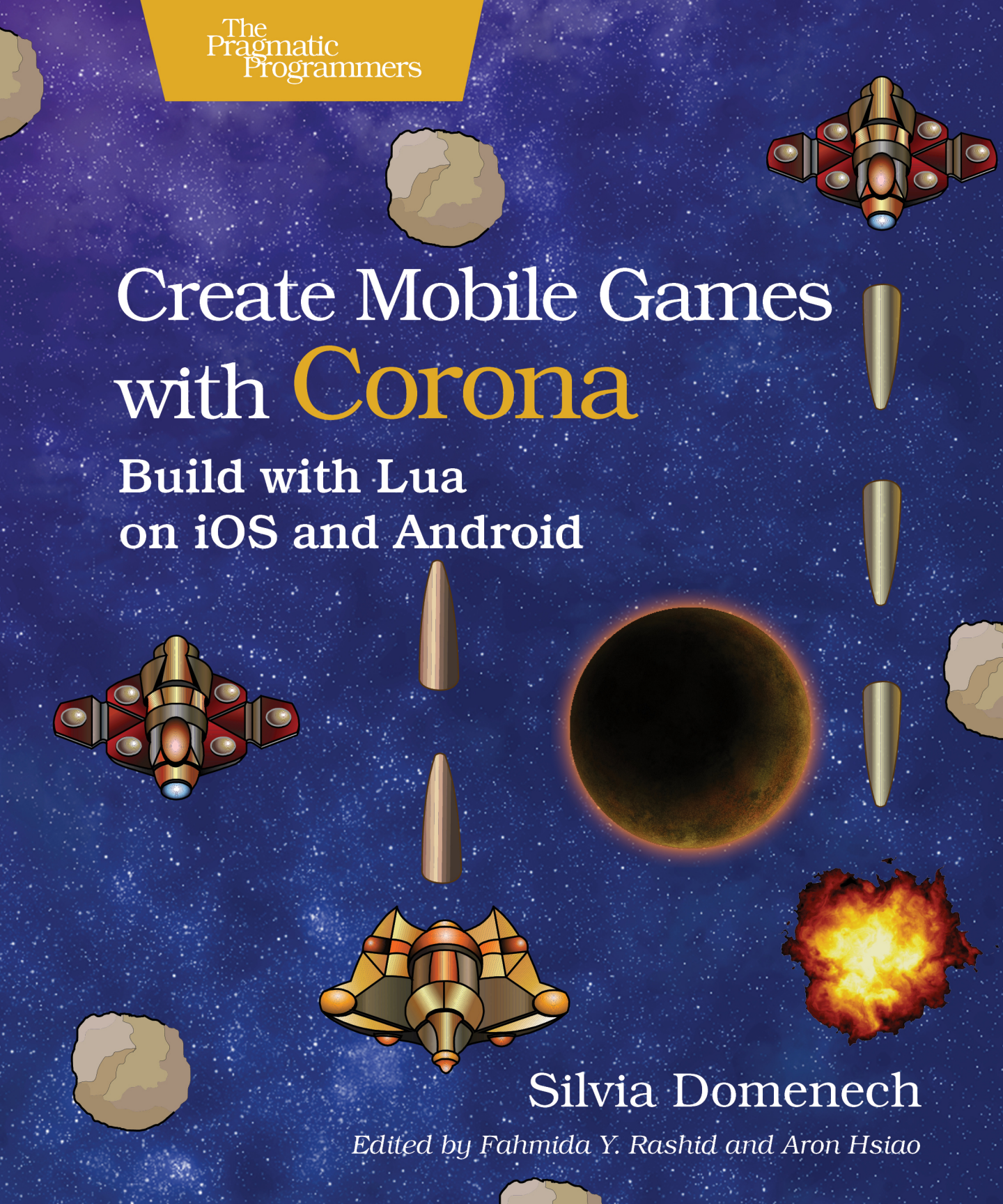
Extracted from:

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

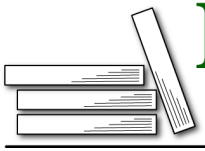
Create Mobile Games with Corona

Build with Lua on iOS and Android

Silvia Domenech

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

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

4.4 Adding Buttons

Even though it's great to add custom tap-based input to interact with units in our games, it's always good to have reusable code for stuff we use frequently, such as buttons. Unless you're making a really simple game, you'll need buttons for the main menu, some of the interfaces, and even online-sharing functions. You can code them from scratch each time you need them, but it's much faster and safer to code them once and reuse them. That way you're sure that the button code works, saving you hours of testing.

When we think of buttons, we think of hit areas, button images, interactivity, and even a pressed state. We need a hit area because we want the user to hit the button easily, even if the button has strange fonts or invisible images. Images are needed in a game because it wouldn't look nice to have text-based buttons, and we need interactivity because a button has to do something to be a button instead of an image.

Corona comes with predefined functions in its Widget API that let us build buttons really quickly. Start by loading the widget class, which lets us build new buttons.

InputAndMenus/menu.lua

```
-- Require the widget class
widget = require "widget"
```

After adding the widget class, you can build a button using a method called widget.newButton(). This function receives several parameters as a Lua table: an id, the button's left and top coordinates, the label text to be displayed, the width and height, and the cornerRadius if you want rounded corners. You also need to pass the onEvent() function you want to get called whenever the user clicks the button.

Build a new Play button in the game's main menu using the widget.newButton() function. Make it a reasonable size, such as 100×30, and place it near the bottom of the screen. Make it call the gotoGame() function we wrote earlier.

InputAndMenus/menu.lua

```
-- Build a "Play" button
local playButton = widget.newButton{
    id = "btnplay",
    label = "Play",
    labelColor = { default={ 0, 0, 0 } },
    left = 100,
    top = 200,
    width = 120,
    height = 40,
    cornerRadius = 10,
    onEvent = gotoGame
```

```
}
group:insert( playButton )
```

I can almost hear you saying, “But that’s a really ugly button!” I agree; a black-and-white button can be fine for serious apps, but it doesn’t look very good in games.

We can use Corona’s sprite sheet system to pass a sprite sheet where we want to get the images from. The button’s default state will show the frame number passed as a variable called defaultFrame, and the pressed state will change this frame to the frame number we passed in overFrame (as shown in the image here). Using these new variables, update the button so that it uses the user-interface (UI) sprite sheet called menuSheets, and frame number 1 as the default state and 2 as the pressed state (see Figure 24, *Play button*).

InputAndMenus/menu.lua

```
function scene:createScene( event )
    local group = self.view
    -- Create a new sprite sheet
    menuSheets = graphics.newImageSheet( "images/menu_buttons.png",
        { width = 120, height = 40, numFrames = 2 } )
    local playButton = widget.newButton{
        -- Make the button use the sprite sheet
        sheet = menuSheets,
        defaultFrame = 1,
        overFrame = 2,
    }
end
```

That’s it! Using an image instead of the default button design was that simple. Don’t you love Corona already? At this point, you can also remove the links on the main menu’s background image. Since we’ve already coded an alternative button, there’s no need to include two ways to start the game.

Figure 24—Play button

4.5 Adding Lives and Difficulty

Right now, the game has lots of enemies, but it will keep going forever. It’s important to limit the game so that it can end within a reasonable timeframe.

We can add a time limit for the level, make it really difficult, or add a progressive difficulty

system. A time limit can be frustrating for good players, because they might like the thrill of playing longer sessions. Increasing the difficulty will make the game too easy for some but too difficult for others, so it's not the ideal choice either.

That leaves us with the option to make the game easier at the beginning and make it harder as time advances. Let's do that and give players a few lives so that a small mistake doesn't kick them out of the game.

Limiting Lives

Limiting the number of lives a player has is really easy. We only need to keep track of a variable, such as lives, and update it each time that the player gains or loses one. In Planet Defender, we'll subtract a life each time an invading ship reaches the central planet, and we won't be adding any lives. The game will end when the player loses all lives.

Let's start by creating the lives variable. Set it to 3, which seems like a reasonable number; players can make a few mistakes, but they won't be forced to play forever.

InputAndMenus/game.lua

```
function scene:enterScene( event )
    -- Start tracking player lives
    lives = 3
end
```

When the game removes a ship that has moved too close to the planet, it means the player didn't destroy it in time. Update the code so that it subtracts a life after removing the spaceship.

Since the game counts the number of remaining lives, it makes sense to check whether the player has run out of lives. If the number of lives left is equal to zero, the game has ended, so call the Storyboard API again to change scenes. If there are no lives left, the game sends players to a game-over scene using storyboard.gotoScene(). (See Figure 25, *Game-over scene*.)

InputAndMenus/game.lua

```
function updateEnemies( )
    for i = #ships, 1, -1 do

        -- Remove dead ships
        if ( ships[ i ]:toggleDelete( ) == true ) then
            -- If the ship was not killed by the player, subtract a life
            if ( ships[ i ].wasKilled == false ) then
                lives = lives - 1
            end
        end
```

```
    end
    -- Check the number of lives
    if lives <= 0 then
        storyboard.gotoScene( "gameover" )
    end
end
```

Figure 25—Game-over scene

Before compiling the app, create a new storyboard scene (using the default Corona template) for the game-over screen. We don't really want to get into too much work here, so just add an image that says "game over" to the scene-creation function. Add an event listener to return to the menu (using addEventListener() and storyboard:gotoScene()). For the gotoMenu() function, you can adapt what we wrote to go to the game in menu.lua.

InputAndMenus/gameover.lua

```
-- Menu listener function. Add a touch listener to the image
function addMenuListener( event )
    bgimage:addEventListener( "touch", gotoMenu )
end

-- Add an image during the scene creation process
function scene:createScene( event )
    local group = self.view

    -- Add a background image to the game over scene
```

```
    bgimage = display.newImage( "images/game_over.jpg" )
    bgimage.anchorX, bgimage.anchorY = 0, 0
    group:insert( bgimage )

end
```

Instead of calling addMenuListener() from the scene creation function, you can wait for a few milliseconds before adding the event listener using Corona's timer.performWithDelay() function. That way, if players tap the game screen a millisecond late, they won't accidentally return to the menu.

InputAndMenus/gameover.lua

```
-- Add a tap event listener to return to the menu
--   but only after some time, to avoid accidental returning
timer.performWithDelay( 500, addMenuListener )
```

The last step is to delete the previous scene. Corona's Storyboard API makes this really easy with the purgeScene() function. You can call this function to force-delete a scene that you've exited. Corona deletes unused scenes automatically in low-memory situations, but this helps your program to consume less unnecessary memory. It also helps if, instead of having to delete scene elements manually, you can delete the scene in one line of code.

Add this call in the game over scene's enterScene() function. It's a good habit to call the purge scene function whenever you don't plan to return to a scene for a while, or if you want to make sure that you have not forgotten to delete any of the scene objects manually. As well as this, purge the menu scene when you enter the game, and purge the game over scene when you go back to the menu.

InputAndMenus/gameover.lua

```
-- If we come from the game, purge the previous scene
storyboard.purgeScene( "game" )
```

It's great to have a lives system in a game, but players will need to know how many lives they have left. Otherwise, losing the game will just feel random. We'll display their lives using a small image in a group.

First create a group where we'll paint the lives and add it to the stage. We're adding the life icons (shown as the heart icon here) to a group to make them easy to move or place them either in front or behind the other objects.

InputAndMenus/game.lua

```
function scene:createScene( event )
    local group = self.view
    -- Add a group to display the lives
    lifeGroup = display.newGroup( )
```

```
    group:insert( lifeGroup )
end
```

In the image sheets in this chapter, the life icon is called heart. Add it three times, updating the x-coordinate with each life so that the images don't all appear in the same location.

InputAndMenus/game.lua

```
function scene:enterScene( event )
    -- Display the lives
    for i = 1, 3 do
        local lifeSprite = display.newImage( imageSheet,
            spritedata:getFrameIndex( IMG_LIFE ) )
        lifeSprite.x = 15 * i - 5
        lifeSprite.y = 10
        lifeGroup:insert( lifeSprite )
    end
end
```

To keep things neat, you can store the image name in a global variable to avoid typing it as a string each time. If you choose that route, remember to define IMG_LIFE in the globals file.

InputAndMenus/globals.lua

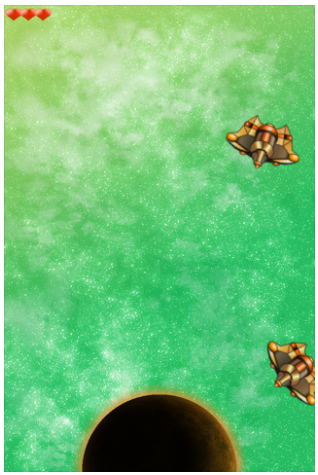
```
-- Icons
IMG_LIFE = "player_life"
```

Since we've just created a set of display objects, we have to remove one each time a life is lost. In this case, we remove the topmost object because it's the rightmost life icon. If you've decided to paint the lives in a different order, just make sure to delete one that makes sense. A UI layout can stop "feeling right" if icons aren't spread out evenly.

Add the call to remove() to the code after you subtract a life from the player.

InputAndMenus/game.lua

```
lifeGroup[ lifeGroup.numChildren ]:removeSelf( )
```



Try to compile the app to check that lives are shown properly and that they're subtracted. Right now, the game will look much more complete (as you can see in the figure here).

Also test whether the game-over screen appears properly—it's easy to make a simple mistake in a scene change, and it's better to spot it before the players do!

Making It More Challenging

Enemies are being added periodically so far, but that's because of the constant difficulty level in the game. We can change difficulty in many ways, but a very easy solution is to increase the number of enemies gradually. That way, inexperienced players will be able to play for a while, and experienced players will have lots of fun when the game becomes too fast even for them.

We can use a logarithmic equation to increase the speed at which we add our ships. To do this, use a shipsAdded variable to store the number of ships we've added to the game.

InputAndMenus/game.lua

```
function scene:enterScene( event )
    -- Create a variable to store the number of ships added
    shipsAdded = 0
end
```

Each time the game adds a ship, add 1 to the variable. Then, update nextEnemy so the next ship can appear after 2000 / (math.log(shipsAdded / 5) + 1) milliseconds. Why logarithmic? That means ships will be added faster each time—but up to a certain limit to avoid making the game ridiculously difficult.

Since Corona works based on frames and a game's frame rate is usually 60 frames per second, there isn't much point in being too accurate when calculating the time required to add the next ship. If you want, update the nextEnemy variable to round it down.

InputAndMenus/game.lua

```
-- Reset the enemy counter
shipsAdded = shipsAdded + 1
nextEnemy = 33 / ( math.log( 1 + shipsAdded / 5 ) + 1 )
```

Voilà! Our game has an incremental difficulty system. It's great for new players because it lets them play for several seconds, but it's also great for veterans because ships will be added faster each time. Since we're using log-

arithms, the speed will increase quickly at the beginning and slowly later. That way, the speed will never increase abruptly, and we won't receive complaints from frustrated players.

4.6 Exercises and Expansion Options

Now that we have finished the chapter, here are some ideas you can try to improve this game.

Power-Ups

In this app, we only add enemies to our game, and the goal is to stop them from reaching our planet. We can add power-ups using the same system as we've used with the ships. Add them randomly after a certain amount of time, and update them in each frame. If players tap them, they can get an extra life. If they don't tap them, then the program needs to remove them eventually. You can do this by adding a frame counter and deleting the power-up if it's inactive after 90 frames (3 seconds) or so.

Splash Screen

Have you noticed splash screens at the start of many games? Developers show their logo for a few seconds before getting into the action. The Storyboard API makes this really easy, because you can use a new scene to show the logo. In the `storyboard.gotoScene()` function, you can pass `"fade"`, `800` as the second (effect name) and third (transition time) parameters to make the scene fade instead of showing an abrupt transition. Most games show a splash screen using `fade`, but you can also play with `SlideDown`, `fromRight`, or any other transition-effect name.

Credits Screen

It's sometimes great to add a credits page to your game just in case somebody wants to check who developed it (and maybe download some of your other games). Since you've learned about scene transitions, it's easy to make a new scene that shows your logo, with a button to go back. Make sure you add your studio name or any games you want to mention there.

4.7 What We Covered

At this point, we know how to use most forms of input for our games. We can now make our units and objects interactive to make apps react to players' actions. Corona also lets us add buttons to make our games easier to use and more intuitive for players, and the Storyboard API is great to make scene transitions in our games without having to code everything manually.

From now on, we'll reuse these tools to make more-complex games. The next app we'll make is a vertical-scrolling shooter, where we'll use everything we've learned from this app and add loading and saving tools, parallax, and explosions.