CHAPTER FIVE

PROGRAMMING

Manual

Table of Contents

1. Introduction to Corona SDK: Easy Cross-Platform Development

Introduction

Corona SDK is an excellent option for any kind of mobile developer from beginner to advanced. This tutorial will introduce you to this easy-to-use cross-platform framework and show you how to start creating content for your preferred platform.

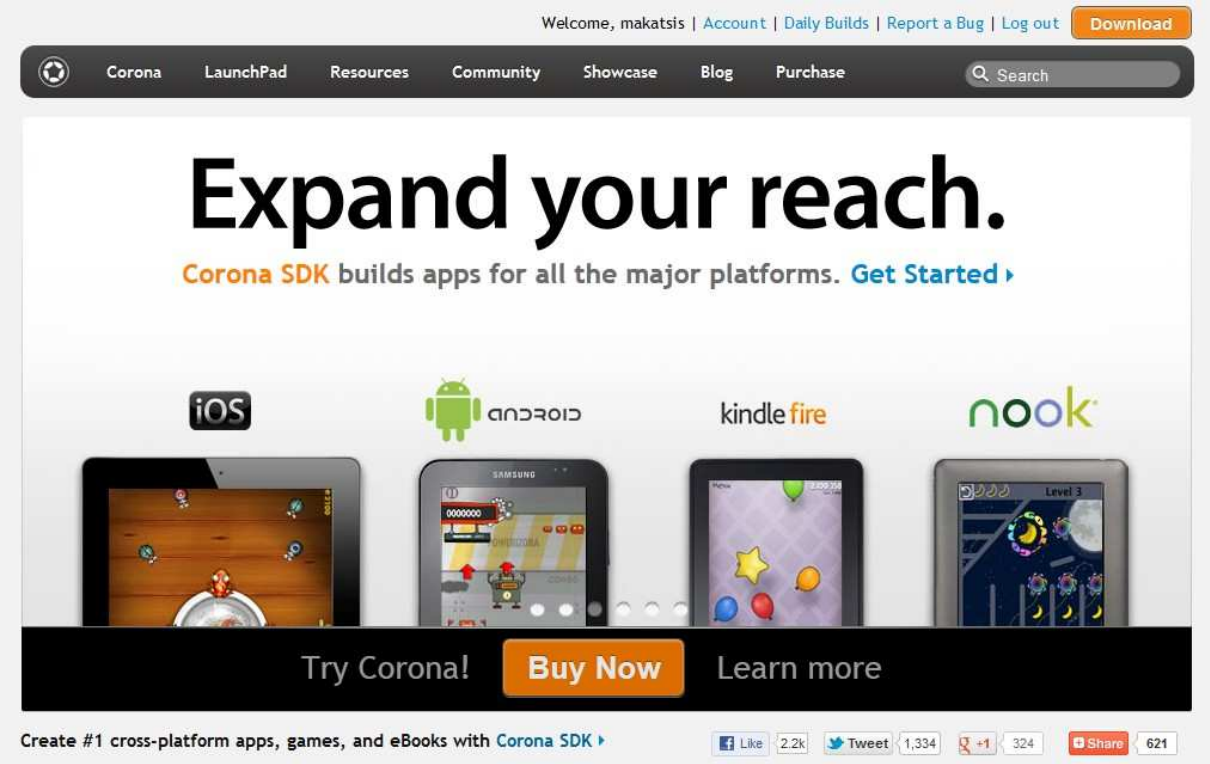


Figure 1: Offial Corona Website

The **official Corona website** describes the SDK as follows:

"Corona is a fast and easy development tool for iPhone, iPad and Android games and applications.

Corona-powered apps run at 30 fps in as little as 300k, and the graphics and animation engine fully leverages OpenGL hardware acceleration.

The **Corona SDK** is the first in Ansca's Corona family of products for creating high-performance multimedia graphically rich applications and games for the iPhone. With Corona, you can quickly create iPhone applications in a matter of hours. No Objective-C/**Cocoa** required, and no **C++**

Ansca is the company behind Corona, and this SDK allows developers to create fast and powerful cross-platform applications that have access to API's other frameworks don't, like the camera, GPS and Accelerometer.

Corona SDK offers plenty of features that make it a very reliable way to create applications. Some of these features are:

- **Native Application Development**: Corona executable binaries are 100% **Objective-C/C++**, so you won't have to worry about Apple iOS 5 new rules on using outside development tools. In fact, Corona needs **Xcode** to compile.

- **Automatic OpenGL-ES Integration**: No need to call extensive classes or functions to create simple screen manipulations.
- **Cross-Platform Development**: Corona can create apps for **iOS** (iPhone, iPod Touch, iPad) and **Android** devices.
- **Performance**: Corona is optimized to make use of hardware-accelerated features, resulting in powerful performance in games and apps.
- **Device Features**: Access device native controls and hardware, like camera, accelerometer, gps, etc.
- **Easy to Learn**: Corona uses the **Lua programming language**, which is powerful and easy to learn.

Supported platforms

The largest advantage of Corona is that it allow you to work with one code base and produce products for many different devices.

Specifically, the **Corona SDK** will allow you to create apps for all **iOS** devices and **Android** devices.

Development with Lua

Figure 2: The Lua Programming Language

Corona uses the **Lua programming language** to create applications. Lua is a scriptitng language commonly used to develop games. It has a good amount of market adoption in the development community. Lua syntax can be compared to languages such as **JavaScript** or **ActionScript 3**, which makes it easy to learn.

Currently, many programming languages are concerned with how to help you write programs with hundreds of thousands of lines. For that, they offer you packages, namespaces, complex type systems, a myriad of constructions, and thousands of documentation pages to be studied.

Lua does not try to help you write programs with hundreds of thousands of lines. Instead, Lua tries to help you solve your problem with only hundreds of lines, or even less. To achieve this aim, Lua relies on extensibility, like many other languages. Unlike most other languages, however, Lua is easily extended not only with software written in Lua itself, but also with software written in other languages, such as **C** and **C++**.

Lua was designed, from the beginning, to be integrated with software written in C and other conventional languages. This duality of languages brings many benefits. Lua is a tiny and simple language, partly because it does not try to do what C is already good for, such as sheer performance, low-level operations, or interface with third-party software. Lua relies on C for those tasks. What Lua does offer is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For that, Lua has a safe environment, automatic memory management, and great facility to handle strings and other kinds of data with dynamic size.

More than being an extensible language, Lua is also a glue language. Lua supports a component-based approach to software development, where we create an application by gluing together existing high-level components. Usually, these components are written in a compiled, statically typed language, such as C or C++; Lua is the glue that we use to compose and connect those components. Usually, the components (or objects) represent more concrete, low-level concepts (such as widgets and data structures) that are not subject to many changes during program development and that take the bulk of the **CPU time** of the final program. Lua gives the final shape of the application, which will probably change a lot during the life cycle of the product. However, unlike other glue technologies, Lua is a full-fledged language as well. Therefore, we can use Lua not only to glue components, but also to adapt and reshape them, or even to create whole new components.

Of course, Lua is not the only scripting language around. There are other languages that you can use for more or less the same purposes, such as **Perl**, **Tcl**, **Ruby**, **Forth**, and **Python**. The following features set Lua apart from these languages; although other languages share some of these features with Lua, no other language offers a similar profile:

Extensibility: Lua's extensibility is so remarkable that many people regard Lua not as a language, but as a kit for building domain-specific languages. Lua has been designed from scratch to be extended, both through Lua code and through external C code. As a proof of concept, it implements most of its own basic functionality through external libraries. It is really easy to interface Lua with C/C++ and other languages, such as **Fortran**, **Java**, **Smalltalk**, **Ada**, and even with other scripting languages.

- **Simplicity**: Lua is a simple and small language. It has few (but powerful) concepts. This simplicity makes Lua easy to learn and contributes for a small implementation. Its complete distribution (source code, manual, plus binaries for some platforms) fits comfortably in a floppy disk.

- **Efficiency**: Lua has a quite efficient implementation. Independent benchmarks show Lua as one of the fastest languages in the realm of scripting (interpreted) languages.

Portability: When we talk about portability, we are not talking about running Lua both on **Windows** and on **Unix** platforms. We are talking about running Lua on all platforms we have ever heard about: **NextStep**, **OS/2**, **PlayStation II (Sony)**, **Mac OS-9** and **OS X**, **BeOS**, **MS-DOS**, **IBM mainframes**, **EPOC**, **PalmOS**, **RISC OS**, plus of course all flavors of Unix and Windows. The source code for each of these platforms is virtually the same. Lua does not use conditional compilation to adapt its code to different machines; instead, it sticks to the standard **ANSI (ISO) C**. That way, usually you do not need to adapt it to a new environment: If you have an ANSI C compiler, you just have to compile Lua, out of the box.

A great part of the power of Lua comes from its libraries. This is not by chance. One of the main strengths of Lua is its extensibility through new types and functions. Many features contribute to this strength. Dynamic typing allows a great degree of polymorphism. Automatic memory management simplifies interfaces, because there is no need to decide who is responsible for allocating and deallocating memory, or how to handle overflows. Higher-order functions and anonymous functions allow a high degree of parametrization, making functions more versatile.

Lua comes with a small set of standard libraries. When installing Lua in a strongly limited environment, such as embedded processors, it may be wise to choose carefully which libraries you need. Moreover, if the limitations are hard, it is easy to go inside the libraries' source code and choose one by one which functions should be kept. Remember, however, that Lua is rather small (even with all standard libraries) and in most systems you can use the whole package without any concerns.

Lua Editors

At this time, Corona doesn't come with an exclusive Lua editor, but there are some great editors already available that you can use:

Free:

- **Eclipse**, using the Lua Eclipse plugin.
- **LuaEdit**, LuaEdit is an IDE/Debugger/Script Editor designed for the version 5.1 of Lua.
- **NotePad++**, a free source code editor which supports several programming languages, including Lua.
- **TextWrangler**, a powerful general purpose text editor and Unix and server administrator's tool.

Commercial:

- **TextMate**, Available for Mac OS X only.
- **BBedit**, a leading professional HTML and text editor for the Macintosh.
- **Decoda**, a professional development environment for debugging Lua scripts in your applications.

Creating your first program

To get started with Corona, let's begin with the classic Hello World application.

Open your prefered Lua editor and write the following code: `print("Hello World!").`

Create a new Project folder named *HelloWorld* and save the file as *main.lua*. We'll launch this app in the next steps.

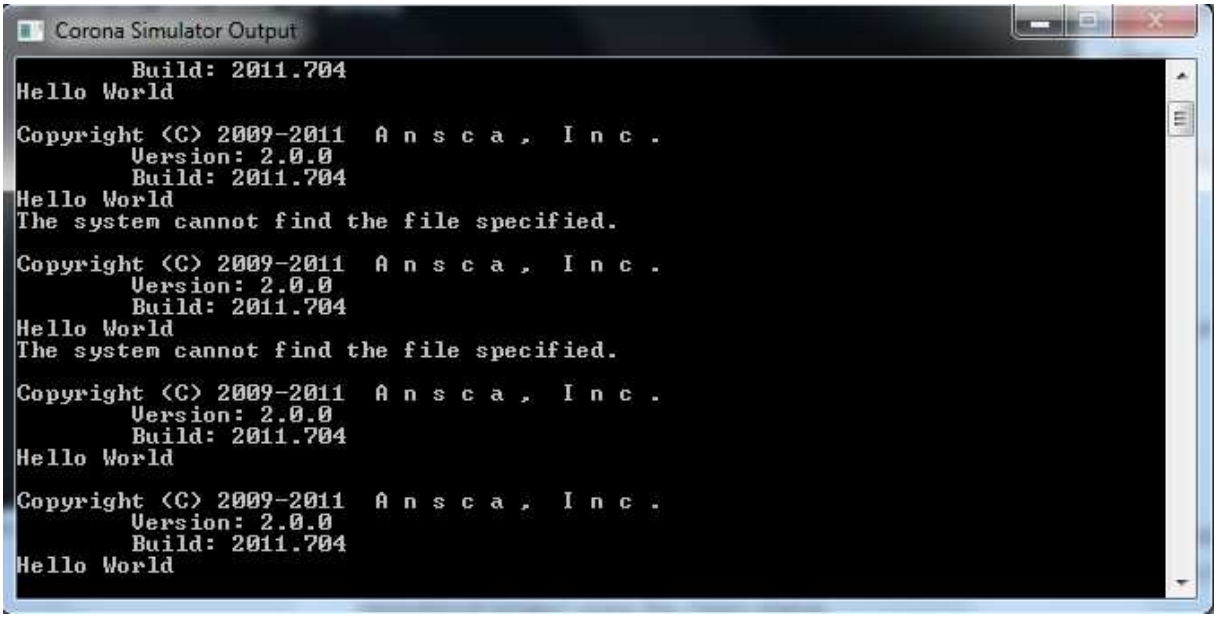


Figure 3: Corona Terminal

This will also open the *Corona Simulator* displaying an iPhone graphic with no content, this is because the print function only outputs to the Terminal, to see how to display text in the simulator continue to the next step.

Simulator

To access the simulator or actual device screen, we'll need to make use of the **Corona specific API's**

In your main.lua file write the following and then run the program again:

```
local myTextField = display.newText("Hello World!", 1, 20, nil, 14);myTextField:setTextColor(255, 255, 255);
```

Figure 4: Corona Simulator

2. CORONA SDK: CREATING AN ANALOG CLOCK APP

Using the Corona API's, we'll create a basic analog clock. The graphics will be PNG's exported from the image editor of your choice and then powered by Lua. You will also learn how to test your application using the simulator and build your app for device testing.

Select Target Device

The first thing you have to do is select the platform you want to run your app, this way you'll be able to choose the size for the images you will use.

The **iOS** platform has these characteristics:

- **iPad**: 1024x768px, 132 ppi
- **iPhone/iPodTouch**: 320x480px, 163 ppi
- **iPhone 4**: 960x640px, 326 ppi

For **Android** it is a little different, being an open platform, you may encounter many different screen resolutions:

- **Nexus One**: 480x800px, 254 ppi
- **Droid**: 854x480px, 265 ppi
- **HTC Legend**: 320x480px, 180 ppi

Interface

Figure 5: Clock Interface

Code

Background

The first thing we'll do is to add the clock background:

```
local background = display.newImage("background.png")
```

This line creates the local variable *background* and uses the display API to add the specified image to the stage. By default, the image is added to 0,0.

Display Clock Hands

We repeat the process with the clock hands and the clock center images, placing them in the center of the stage:

```
local hourHand = display.newImage("hourHand.png", 152, 185)
local minuteHand = display.newImage("minuteHand.png", 152, 158)
local center = display.newImage("center.png", 150, 230)
local secondHand = display.newImage("secondHand.png", 160, 155)
```

Reference Point

To position the images correctly, we modify the reference point in order to move images relatively to the bottom center:

```
hourHand:setReferencePoint(display.BottomCenterReferencePoint)
minuteHand:setReferencePoint(display.BottomCenterReferencePoint)
secondHand:setReferencePoint(display.BottomCenterReferencePoint)
```

Initial Position

Here we set the initial position of the clock hands. This time we set the rotation according to the system time:

```
local timeTable = os.date("*t")

hourHand.rotation = timeTable.hour * 30 + (timeTable.min * 0.5)
minuteHand.rotation = timeTable.min * 6
secondHand.rotation = timeTable.sec * 6
```

Memory Practices

The *timeTable* variable will be used just once at the application launch, so there's no need to keep it in memory. To release the memory used by the variable (which is almost nothing, but you MUST get used to deallocate unused vars or objects) we set its value to *nil*, this way garbage collection takes care of it:

```
timeTable = nil
```

MoveHands Function

The next lines of code handle the clock hands rotation, it is the same code as before, only this time wrapped into a function that will be executed every second by a *Timer*:

```
local function moveHands(e)
	local timeTable = os.date("*t")
	hourHand.rotation = timeTable.hour * 30 + (timeTable.min * 0.5)
	minuteHand.rotation = timeTable.min * 6
	secondHand.rotation = timeTable.sec * 6
end
```

Timer

The *Timer*, it executes every second and performs the specified function, this is the *moveHands* function we created in the last step. The times it's executed are set by the third parameter, 0 is infinity.

Figure 6: Corona Simulator analogue Clock

3. Accelerometer Application Overview

Using the **Corona API's**, we'll create a basic application that registers the device movement based on the accelerometer value, moving an object on the screen.

Select Target Device

The first thing you have to do is select the platform you want to run your app, this way you'll be able to choose the size for the images you will use.

The **iOS** platform has these characteristics:

- **iPad**: 1024x768px, 132 ppi
- **iPhone/iPodTouch**: 320x480px, 163 ppi
- **iPhone 4**: 960x640px, 326 ppi

For **Android** it is a little different, being an open platform, you may encounter many different screen resolutions:

- **Nexus One**: 480x800px, 254 ppi
- **Droid**: 854x480px, 265 ppi
- **HTC Legend**: 320x480px, 180 ppi

Interface

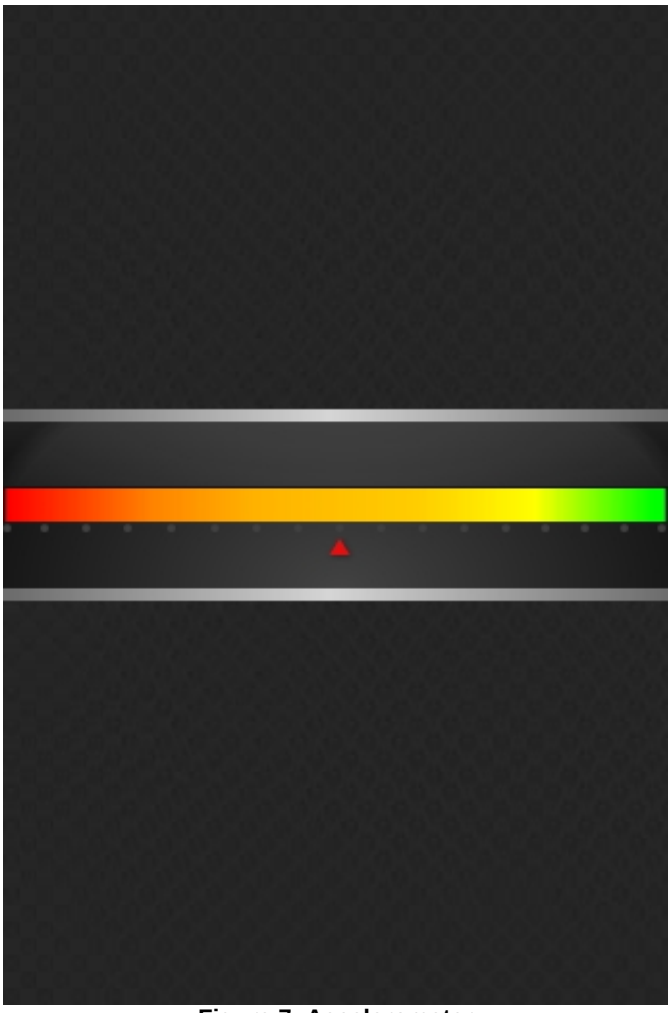


Figure 7: Accelerometer

This is the graphic interface we'll be using, it includes a triangle graphic that will serve as the position meter.

Code

Hide Status Bar

First, we hide the status bar, this is the bar on top of the screen that shows the time, signal, and other indicators

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

Background

Now we add the app background.

```
local background = display.newImage("background.png")
```

This line creates the local variable *background* and uses the *display* API to add the specified image to the stage. By default, the image is added to 0,0 using the top left corner as the reference point.

Indicator

We repeat the process with the position indicator image, placing it in the center of the stage.

```
local indicator = display.newImage("indicator.png")

indicator:setReferencePoint(display.CenterReferencePoint)
indicator.x = display.contentWidth * 0.5
indicator.y = display.contentWidth * 0.5 + 100
```

Needed Variables

The next variables will be used to handle the accelerometer event.

- **acc**: A Table that will be used as a function listener for the accelerometer event.
- **centerX**: Stores the horizontal center value of the stage.

```
local acc = {}
local centerX = display.contentWidth * 0.5
```

Accelerometer Function

This function uses the *acc* table to create a listener for the accelerometer event, the *xGravity* property (part of the accelerometer event) and the centerX variable moves the position indicator according to the calculated position.

```
function acc:accelerometer(e)
	indicator.x = centerX + (centerX * e.xGravity)
end
```

This will make our indicator to balance when the device inclination changes, the *xGravity* property will handle the side movements, you can use the *yGravity* property to handle up/down inclination types.

Accelerometer Listener

The Accelerometer events are runtime based, so we use the Runtime keyword to add the listener.

```
Runtime:addEventListener("accelerometer", acc)
```

4. Develop an Entertaining Magic Ball Application

Using the *Shake* Event built in the **Corona API**, we'll create an application that generates a random result from predefined words.You'll also learn to create simple animations using the *transition* methods.

Select Target Device

The first thing you have to do is select the platform you want to run your app, this way you'll be able to choose the size for the images you will use.

The **iOS** platform has these characteristics:

- **iPad**: 1024x768px, 132 ppi
- **iPhone/iPodTouch**: 320x480px, 163 ppi
- **iPhone 4**: 960x640px, 326 ppi

For **Android** it is a little different, being an open platform, you may encounter many different screen resolutions:

- **Nexus One**: 480x800px, 254 ppi
- **Droid**: 854x480px, 265 ppi
- **HTC Legend**: 320x480px, 180 ppi

Interface

Figure 8: Magic Ball

This is the graphic interface we'll be using, it includes a triangle graphic that will serve as the Octohedron found in Magic Balls.

Code

First, we hide the status bar, this is the bar on top of the screen that shows the time, signal and other indicators.

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

Background

Now we add the app background.

```
local background = display.newImage("background.png")
```

This line creates the local variable *background* and uses the *display* API to add the specified image to the stage. By default, the image is added to 0,0 using the top left corner as the reference point.

Octohedron

We repeat the process with the octohedron image, placing it in the center of the stage.

```
local octohedron = display.newImage("octohedron.png", 110, 186)
octohedron.isVisible = false
```

The Octothedron will be invisible by default, and will appear at the first device shake.

TextField

The following code creates the center TextField that will display the random sentence when a shake event is dispatched.

```
local textfield = display.newText("", 0, 0, native.systemFontBold, 14)

textfield:setReferencePoint(display.CenterReferencePoint)
textfield.x = display.contentWidth * 0.5
textfield.y = display.contentHeight * 0.5
textfield:setTextColor(255, 255, 255)
```

Necesary Variables

The next variables will be used to handle the Shake event.

- **shake**: A Table that will be used as a function listener for the shake event.
- **options**: Stores the words that can be shown by the magic ball.

```
local shake = {}
```

```
local options = {"Probably Not", "No.", "Nope", "Maybe", "Yes", "Probably", "It's Done", "Of Course"}
```

Shake Function

This function listens for a shake event and reveals the octohedron and text if true.

```
function shake:accelerometer(e)
	if(e.isShake == true) then
		octohedron.isVisible = true
		transition.from(octohedron, {alpha = 0}) -- Show octohedron
		textfield.text = options[math.random(1, 8)]
		transition.from(textfield, {alpha = 0})
	end
end
```

Accelerometer Listener

The Accelerometer events are runtime based, so we use the Runtime keyword to add the listener.

```
Runtime:addEventListener("accelerometer", shake)
```

Figure 9: Magic Ball - Final Result

4. WORKING WITH ALERTS

Alerts are a predefined system method to show information to the user, they are commonly used to display short messages and can include one or multiple options to determine a posterior action.

Select Target Device

The first thing you have to do is select the platform you want to run your app, this way you'll be able to choose the size for the images you will use.

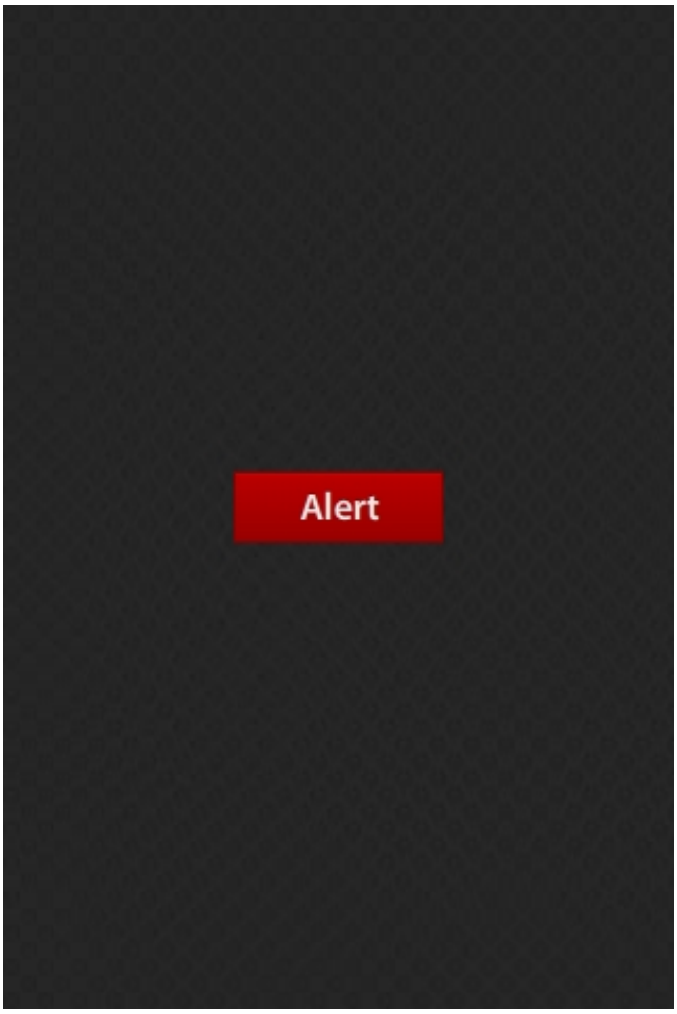
The **iOS** platform has these characteristics:

- **iPad**: 1024x768px, 132 ppi
- **iPhone/iPodTouch**: 320x480px, 163 ppi
- **iPhone 4**: 960x640px, 326 ppi

For **Android** it is a little different, being an open platform, you may encounter many different screen resolutions:

- **Nexus One**: 480x800px, 254 ppi
- **Droid**: 854x480px, 265 ppi
- **HTC Legend**: 320x480px, 180 ppi

Interface



Code

Hide Status Bar

First, we hide the status bar, this is the bar on top of the screen that shows the time, signal and other indicators.

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

Background

This line creates the local variable *background* and uses the *display* API to add the specified image to the stage. By default, the image is added to 0,0 using the top left corner as the reference point.

```
local background = display.newImage("background.png")
```

Alert Button

Repeat the process with the button image, placing it in the center of the stage. The button function will be created later in the code.

```
local alertButton = display.newImage("alertButton.png")

alertButton:setReferencePoint(display.CenterReferencePoint)
alertButton.x = 160
alertButton.y = 240
```

Lister for Alert Clicks

When the user clicks on any of the option buttons in the Alert a *clicked* event is displatched, we need to check for the *index* of the clicked button in order to know which option was selected. An alert lets you include up to 6 buttons, its index is defined by the order it was written in the alert call.

```
local function onClick(e)
	if e.action == "clicked" then
		if e.index == 1 then
			-- //Some Action
			elseif e.index == 2 then
				system.openURL( "http://www.ebusiness-lab.gr" )
		end
	end
end
```

Display Alert

This function will be executed when the alert button is pressed, it uses the *native.showAlert()* method to display the alert. The alert will be linked to a variable that will serve as the alert ID, this way it can be located, reused or removed by the *native.cancelAlert()* method.

```
function alertButton:tap(e)
	local alert = native.showAlert("TEI Messolonghi", "Mobile Development at TEI Mesolonghi", {"OK", "Learn More"}, onClick)
end
```

This method has four paremeters, lets take a look at them:

native.showAlert(**title**, **message**, {**buttons**}, **listener**)

- **title**: The text on top of the alert.
- **message**: The body of the alert.
- **buttons**: A table containing the buttons that will be displayed by the alert, you can display up to 6 buttons.
- **listener**: A function that will listen to the alert button's click events.

Alert Button Listener

The button now has a function to run when pressed, but this function alone will not be able to react without a listener.

The next line of code sets that listener:

```
alertButton:addEventListener("tap", alertButton)
```

Figure 10: Alert message – Simulation

5. Creating a Simple Basketball Game With Corona

The physics engine that comes with Corona Game Edition is an incredibly powerful and easy to use tool. In this tutorial, we will cover the completion of a rudimentary basketball game using this exciting technology.

Figure 11: The final project, running on the iPhone simulator

Step 1: Setting Up the Physics Engine

```
1. display.setStatusBar(display.HiddenStatusBar)
2.
3. local physics = require "physics"
4. physics.start()
5. physics.setGravity(9.81, 0)  -- 9.81 m/s*s in the positive x direction
6. physics.setScale(80)  -- 80 pixels per meter
7. physics.setDrawMode("normal")
```

The first thing we do (as in many programs) is get rid of the status bar at the top of the screen. Next, we make the necessary require statement for using physics and store the result in the aptly named “physics” variable. Things become more interesting in the next few lines. In line five, we set the gravitational acceleration. Typically, gravity is set to 9.8 m/s*s in the positive y-direction, but in this instance we want to make gravity pull in the positive x-direction because the application will have a landscape orientation. Furthermore, we set the scale to 80 pixels per meter. This number can vary quite a bit based on the size of the objects in your application, and you may have to play around with it to give your game the correct feel. I chose 80 px/m because I want to fit about 15 feet of vertical space on the screen. Knowing that, it’s just a simple matter of unit conversion to get a value.

Note: It’s important to try and tie everything to real world objects in applications with physics. The more real life measurements you use, the less guesswork there will be and the more realistic your application will seem.

We round out these few lines by setting the draw mode to normal. This line makes it easier to change to debug mode later if we should have to fix some unintended behavior with collisions. Setting this to normal is the default behavior and draws the shapes as the user will see them in the final game.

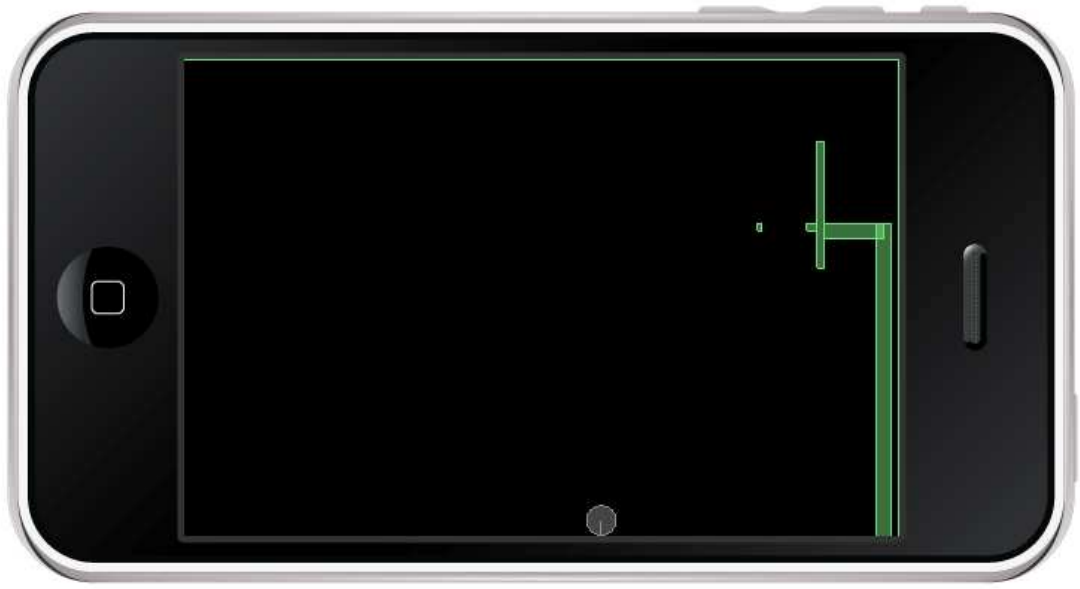


Figure 12: The final project viewed in "debug" mode

Step 2: Creating the Arena

```
1.  local background = display.newRect(0,0,display.contentWidth,display.contentHe
    ight)
2.  local score = display.newText("Score: 0", 50, 300)
3.  score:setTextColor(0, 0, 0)
4.  score.rotation = -90
5.  score.size = 36
6.  local floor = display.newRect(320, 0, 1, 480)
7.  local lWall = display.newRect(0, 480, 320, 1)
8.  local rWall = display.newRect(0, -1, 320, 1)
9.  local ceiling = display.newRect(-1, 0, 1, 480)
10.
11. staticMaterial = {density=2, friction=.3, bounce=.4}
12. physics.addBody(floor, "static", staticMaterial)
13. physics.addBody(lWall, "static", staticMaterial)
14. physics.addBody(rWall, "static", staticMaterial)
15. physics.addBody(ceiling, "static", staticMaterial)
```

This block establishes the boundaries of the arena, and the properties of all the static objects in the application. We begin by adding a simple (white by default) rectangle to the background. Inside of the white rectangle in the background image, we position some text to display the current score. Because the application will be displayed in landscape mode, we also make necessary rotation adjustments here. The arena needs to trap the ball within the visible portion of the screen. We achieve this with four static rectangles (floor, lWall, rWall, ceiling) placed just out of view.

Next, we bring physics back into the equation. Instead of retyping the table for the physical properties of each object, we create a table name staticMaterial to be reused for each of the walls and the goal itself. I've chosen fairly standard values for these properties, though I encourage you to play around with them. There is one more step we must take, and that is to tell Corona that these objects should participate in physics calculations. We do this by calling the addBody function of the physics object. This function takes three arguments:

1. The object
2. An optional modifier
3. A table of physical properties

We've already determined the properties and the objects, so all that remains is the optional modifier. We use "static" to prevent gravity, or any force for that matter, from displacing our walls!

Figure 13: The white background, the score table & invisible walls

Step 3: Adding a Ball and a Goal

```
1.  -- Create the goal
2.  local vertPost = display.newRect(110, 5, 210, 10)
3.  vertPost:setFillColor(33, 33, 33)
4.  local horizPost = display.newRect(110, 10, 10, 40)
5.  horizPost:setFillColor(33, 33, 33)
6.  local backboard = display.newRect(55, 50, 85, 5)
7.  backboard:setFillColor(33, 33, 33)
8.
9.  physics.addBody(vertPost, "static", staticMaterial)
10. physics.addBody(horizPost, "static", staticMaterial)
11. physics.addBody(backboard, "static", staticMaterial)
12.
13. --Create the Ball
14. local ball = display.newCircle(50, 200, 10)
15. ball:setFillColor(192, 99, 55)
16.
17. physics.addBody(ball, {density=.8, friction=.3, bounce=.6, radius=10})
```

In one fell swoop, we create the rest of the visual elements of our app. This should all look very familiar. There are just two things that I would like to point out. First, some of the values for the positioning of the goal may seem off. This is to account for the landscape orientation. The goal will appear upright when the device is rotated on its side. Also, be sure to include the radius property in the properties table of the ball so it will behave correctly.

Figure 14: After adding a ball and a goal

Step 4: Creating Drag Support for the Ball

```
1.  local function drag( event )  -- create a function to execute when the ball
    is touched
2.      local myball = event.target   -- event.target is "who" is capturing the
    event
3.
4.      local phase = event.phase -- event.phase describes the touch sequence:
    "began", "moved", "canceled", etc...
5.      if "began" == phase then
6.          display.getCurrentStage():setFocus( myball )  -- by setting focus to
    the ball we instruct the system to deliver all future hit events to the same
    object, so that we can drag the ball around (the "moved" phase, below)
7.
8.          -- store initial position: we store the horizontal & vertical
    difference between the ball and the touch positions, so that we can drag the
    ball from anywhere on it's surface
9.          myball.x0 = event.x - myball.x
10.         myball.y0 = event.y - myball.y
11.
12.         -- avoid gravitational forces
13.         event.target.bodyType = "kinematic"
14.
15.         -- stop current motion, if any
16.         event.target:setLinearVelocity( 0, 0 )
17.         event.target.angularVelocity = 0
18.
19.     else
20.         if "moved" == phase then
21.             myball.x = event.x - myball.x0
22.             myball.y = event.y - myball.y0
23.         elseif "ended" == phase or "cancelled" == phase then
24.             display.getCurrentStage():setFocus( nil )  -- clear focus, user
    can touch any other objects after this
25.             event.target.bodyType = "dynamic"  -- re-enable gravity
26.         end
27.     end
```

```
28.
29.     return true  -- when an event handler returns true, no other handlers get executed after this one
30. end
31. myball:addEventListener("touch", drag) -- make ball listen to the touch event and reply with the drag function
```

This function gives us very basic drag support. Some of the high points include setting the bodyType of the ball to kinematic so gravity won't pull the ball out of the user's hands (**Note: be sure to set this back to dynamic after the touch has ended**). The lines just after that are equally important. There we stop all of the ball's motion when it is touched to avoid the same problem we had with gravity.

If you run the app as it is now, you will probably notice that the ball loses all of its momentum as soon as you stop touching it. To remedy this, we need to create a function to track the speed of the ball, and then set the speed of the ball appropriately after the touch ends.

```
1.  local speedX = 0
2.  local speedY = 0
3.  local prevTime = 0
4.  local prevX = 0
5.  local prevY = 0
6.
7.  function trackVelocity(event)
8.      local timePassed = event.time - prevTime  -- time is given in msec
9.      prevTime = prevTime + timePassed
10.
11.     speedX = (myball.x - prevX)/(timePassed/1000)  -- velocity is counted at pixels/sec
12.     speedY = (myball.y - prevY)/(timePassed/1000)
13.
14.     prevX = myball.x
15.     prevY = myball.y
16. end
17.
18. Runtime:addEventListener("enterFrame", trackVelocity)  -- trackVelocity gets executed everytime the screen is redrawn
```

We create trackVelocity as a listener of the enterFrame event, so it is called everytime the screen is redrawn. What it does is find the change in speed over the change in time to find the velocity of the ball in pixels per second. There's really not much to it. Add the following line to the drag function to properly set the linear velocity of the ball.

```
1.  myball:setLinearVelocity(speedX, speedY) -- when myball is released it gets the computed velocity
```

Step 5: Creating the Hoop and Scoring Mechanism

We begin with some more visual work, but by now you should be a pro at rectangles, so it should be painless. The following code creates the rim. Notice that the middle portion of the rim is not going to be part of the physical system because we want the ball to pass through freely.

```
1.  local rimBack = display.newRect(110, 55, 5, 7)
2.  rimBack:setFillColor(207, 67, 4)
3.  local rimFront = display.newRect(110, 92, 5, 3)
4.  rimFront:setFillColor(207, 67, 4)
5.  local rimMiddle = display.newRect(110, 62, 5, 30)
6.  rimMiddle:setFillColor(207, 67, 4)
7.
8.  physics.addBody(rimBack, "static", staticMaterial)
9.  physics.addBody(rimFront, "static", staticMaterial)  -- both the back and front of the rim should have a static body
```

Next we need a way to know when the ball has passed through the goal. The easiest way to accomplish this is by designating a small patch of the screen near the rim as a "score zone".

Whenever the ball is in this zone we can increment the score. To prevent the score from miscounting when the ball lingers around the rim, we keep track of the time of the last goal, and ensure that there is adequate separation between each successive goal. A one second delay should work nicely.

```
1. scoreCtr = 0
2. local lastGoalTime = 1000
3.
4. function monitorScore(event)
5.     if event.time - lastGoalTime > 1000  then  -- allow execution only after
   1 second
6.         if ball.x > 103 and ball.x < 117 and ball.y > 62 and ball.y < 92 then

7.             scoreCtr = scoreCtr + 1
8.             print(score.text)
9.             lastGoalTime = event.time
10.            score.text = "Score: " .. scoreCtr
11.        end
12.    end
13. end
14. Runtime:addEventListener("enterFrame", monitorScore)  -- scoring is monitored
    everytime the screen is redrawn
```

Conclusion

Corona takes care of the more difficult physics tasks, leaving you with more time to focus on the content and gameplay of your game.

(Reference: Carter Grove, November 2010, mobile tuts+ http://mobile.tutsplus.com/tutorials/corona/corona-sdk_game-development_basketball)

Basketball Game - Full Code

Included external file (main.lua)