
Corona™ SDK

Applications Programming Guide





Anasca Inc.
© 2009 Anasca Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Anasca Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Anasca's copyright notice.

The Anasca logo is a trademark of Anasca Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Anasca retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications using Anasca software. Every effort has been made to ensure that the information in this document is accurate. Anasca is not responsible for typographical errors.

Even though Anasca has reviewed this document, ANSCA MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL ANSCA BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Anasca dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Lua 5.1 Copyright © 1994-2008 Lua.org, PUC-Rio.

iPhone™, App Store™ and Mac OS® are trademarks of Apple, Inc.

OpenGL® ES is a trademark of Khronos Group.

Photoshop® and Illustrator® are registered trademarks of Adobe, Inc.

Anasca™, Corona™ and Corona SDK™ are trademark of Anasca, Inc.

Contents

Preface	8
Limitations	8
See Also	8
Tutorial Introduction	9
Hello World	9
Simulator vs Terminal	10
Hello World on the Simulator	10
Rapid Prototyping	11
Basic Interactivity	12
Animation and Sound	13
Projects	14
Assets (Building Blocks)	14
Starting a Project	14
Building for Device	15
Sample Code	16
Application Environment	18
Life Cycle	18
Global Runtime Object	18
Sandbox	18
Application Events	18
Termination	18

Interruptions	19
Customization	19
Application Icon	19
Launch Image	19
Event Handling	20
<hr/>	
Global Events	20
Local Events	20
Hit Events	20
Propagation and Handling of Events	20
Overriding Propagation with Focus	21
Listeners and Event Delivery	21
Registering for Events	22
Conventions	22
Graphics and Drawing	23
<hr/>	
Creating Display Objects	23
Painter's Model	23
Display Hierarchy	23
Group Objects	23
Stage Objects	24
Moving Objects Forward and Backward	24
Drawing Cycle	25
Screen Updates	25
Coordinates and Transforms	25
Coordinates	25
Changing Position of Objects	26
Transforms	26

Object References	27
Removing Objects Properly	27
Variable References	28
Common Pitfalls	29
Animation	31
<hr/>	
Basic Animations	31
Animated Sprites (or “Movieclips”)	32
Custom/Programmatic Animations	32
Frame Rate	33
Time-based vs Frame-based	33
Lost or Missing Time	34
User Interface	36
<hr/>	
Buttons	36
Alerts	36
Text Input	36
Files	37
<hr/>	
Getting Paths to Files	37
Reading Files	37
Writing Files	38
Beware Security Violations	38
Networking	39
<hr/>	
Downloading Files	39
Uploading Files	39
Multimedia	40
<hr/>	

Playing Sound	40
Event Sound	40
Longer Sounds	40
Playing Video	40
Device Support	42
<hr/>	
Orientation	42
Camera	42
Common Design Tasks	43
<hr/>	
Strategies to Avoid Globals	43
Overcoming Scoping Issues of Locals	43
Keeping Everything Local in Listeners	43
Pausing and Restarting Animations	43
Managing Screens	43
Saving Data on Application Exit	44
Restoring Data on Application Launch	44
Performance and Optimization	46
<hr/>	
Using Memory Efficiently	46
Example	46
Reducing Power Consumption	47
Network	48
CPU	48
Graphics	48
Group objects	48
Turn off animations for non-visible objects	48
Optimize image sizes	49
Minimize setup code at startup time	49

Lua: Best Practices	49
Use locals (i.e. avoid global variables)	49
Math: fast vs slow	50
Inserting objects into arrays	50
Constant Folding	50
Cache properties in a local variable	50
Tuning Your Code	51
Revision History	52

Preface

Ansca™ Corona™ will fundamentally change how you approach iPhone™ software development whether you're an engineer, a web developer, or a designer.

The Corona SDK™ allows you to create native apps for the iPhone. These apps look and behave as normal applications natively built for a device. You get full access to device-specific features such as the camera or accelerometer. Your application will automatically leverage the performance benefits of being a native executable, especially hardware-accelerated sound and graphics.

The process is simple:

- Design images, audio, video, and animation assets using your favorite creative tool.
- Rapidly develop your iPhone apps using the Corona SDK taking full advantage of the device including accelerometer, touch screen, OpenGL®ES, and more.
- Build native optimized apps and distribute them on the iPhone App Store™.

As a developer, you will program in Lua, a simple and intuitive scripting language with exceptional performance, and leverage Corona's innovative and robust APIs.

This document will discuss how to use Corona's APIs to maximize your productivity.

Limitations

Currently, the tools only work on Mac OS® X 10.5.6 or later. Also, an Intel Mac is required to create iPhone device builds, due to Apple's code-signing requirements.

See Also

- *Corona SDK Language and API Reference* provides reference information for all functionality offered by Corona.
- A highly recommended detailed and authoritative introduction to all aspects of Lua programming by Lua's chief architect: **Programming in Lua** (2nd edition), by Roberto Ierusalimsky
- For an official definition of the language, consult **Lua 5.1 Reference Manual**, by R. Ierusalimsky, L. H. de Figueiredo, W. Celes. Also available online at <http://www.lua.org/manual/5.1/>
- Additional documentation is available at <http://www.lua.org/docs.html>
- A live, interactive demo of Lua is available at <http://www.lua.org/demo.html>. This is an excellent place to see some simple sample programs and to play with your own.

Tutorial Introduction

Let's start with a quick introduction to the Corona SDK. We'll focus just on the essentials without getting stuck in the details. We're not trying to be complete or precise. Rather, we want to get you as quickly as possible to the point where you can start creating cool, useful, or engaging apps.

We'll talk about some basics like variables and functions as well as incorporate animation and interactivity so you get a sense of what's possible. Keep in mind this is just a tutorial, so you won't find a complete explanation of any one feature.

Our aim is to address multiple audiences. Experienced programmers should be able to extrapolate the information in this chapter for their own needs. Beginners can use this as a springboard for writing their own small, simple apps.

Hello World

The best (only) way to learn how to use the Corona SDK is by writing an app. To do that we write programs in a language called Lua. In keeping with tradition, let's write a some Lua code that prints "Hello World".

The big roadblock, here, is figuring out the details of how to actually use the Corona SDK to do this. Once you've got these details mastered, everything gets a lot easier.

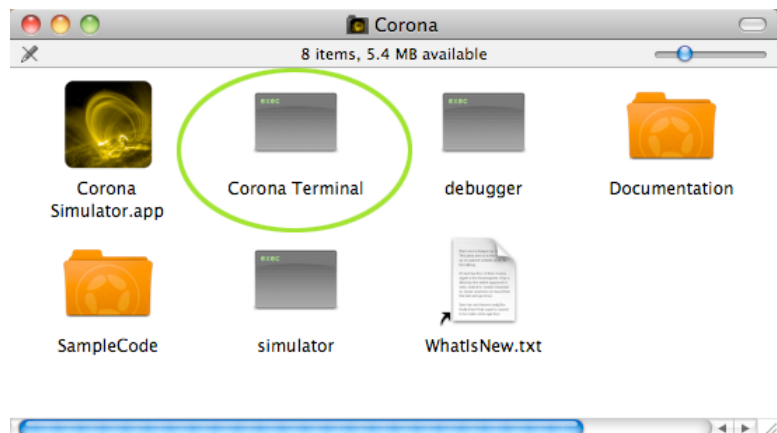
So let's get started! What you'll need is a text editor to write your program in. Later, you'll save that file to a folder so that the Corona Simulator can run and show you the results.

In the text editor, type the following:

```
print( "Hello World" )
```

Then save it to a file called `main.lua` in some folder that's easy to locate. Generally, every program should have its own folder on your system.

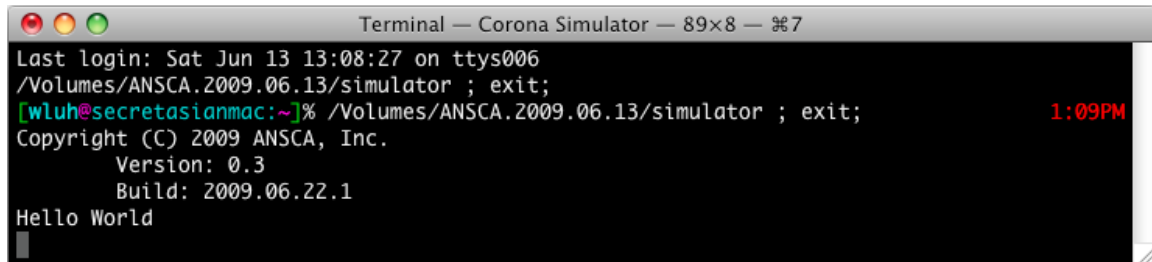
To run the program, you need to launch the Corona Simulator. All Corona SDK files should be in the Corona folder in your Applications folder (see "Getting Started Guide" for more information on how to install the Corona SDK.) The contents of the SDK will look something like the picture at right.



Double-click on the icon for Corona Terminal (the circled icon at right).

This will launch a Terminal window and bring up a file dialog. In the dialog, navigate to the folder containing your `main.lua` file and click the **Open** button.

At this point, you will see “Hello world” in the Terminal window:



```
Terminal - Corona Simulator - 89x8 - %7
Last login: Sat Jun 13 13:08:27 on ttys006
/Volumes/ANSCA.2009.06.13/simulator ; exit;
[wluh@secretasianmac:~]% /Volumes/ANSCA.2009.06.13/simulator ; exit;
Copyright (C) 2009 ANSCA, Inc.
Version: 0.3
Build: 2009.06.22.1
Hello World
```

You will also see a blank simulator window (right) that simulates what would display on the actual phone. In this case, the phone screen remains blank because we’re told the program to output to the Terminal.



Let’s explain how this program worked. The app launches from the file called `main.lua`. The simulator loads this file and follows the instructions contained inside. Generally, an app consists of *statements* and *variables*. Statements provide instructions on what operations and computations need to be done; variables store the values of these computations.

In this program, we use a *function* called `print`. A function is just a collection of statements that perform some task. You send inputs into the function called *parameters* (or *arguments*). Some functions return results. In the case of `print`, all it does is output the arguments as strings to the Terminal.

Simulator vs Terminal

So why did “Hello world” only display in the Terminal window and not in the simulator? That’s because `print` is designed to output messages to the Terminal. Its purpose is to output you to send diagnostic messages about what’s happening in your program. And in general, the Terminal window gives you the ability to see warning/error messages that the simulator generates or print your own messages.

Hello World on the Simulator

To get things displaying on the simulator screen, we need to use different functions that come from Corona’s graphics *library*. A library is a collection of functions that provides useful, but

related functionality. To show "Hello World" in the simulator, you need to add the following two lines:

```
local textObject = display.newText( "Hello World!", 50, 50, nil, 24 )
textObject:setTextColor( 255,255,255 )
```

Let's explain what's happening. `newText` is a function of the `display` library that returns an object that will represent the text on the screen.

Library functions deserve some more explanation. In this example, you can think of `newText` as belonging to the `display` library. This relationship is known as a *property* relationship. So to access the `newText` property of `display`, you have to use a dot. Hence, you write "`display.newText`" but not "`newText`" by itself.

The function `setTextColor` is a special method called an *object method*. It uses a special colon syntax that means it is related to the text object you created. Typically these methods modify the variable before the colon (i.e. `textObject`). In this case, the text object has no color by default, so we need to assign the Red, Green, and Blue color channels. These numbers range from 0-255. So for white, we need 255 for each channel.



Rapid Prototyping

One of the most powerful things about the Corona SDK is the ability to make quick changes and see those changes instantly.

Let's revisit our previous example to see how this works. You can also start with the "HelloWorld" project in the Sample Code. Launch the simulator, navigate to the folder for your program, and click open as you did before.

Now, open up the `main.lua` file in your text editor and try changing the 3 arguments to `setTextColor`. For example, you might have done something like:

```
local textObject = display.newText( "Hello World!", 50, 50, nil, 24 )
textObject:setTextColor( 255, 0, 0 )
```

Now, *save* the file and then go back to the simulator. Under the **File** menu, click the **Relaunch** submenu item (**File** -> **Relaunch**) — or use the keyboard shortcut ⌘R (command-R) in the simulator. This reloads your `main.lua` file without having to restart the simulator. Notice how the color of the text changes immediately. In the version above, the text would appear red.

As you develop your application, you'll find yourself doing this sort of thing very often. You'll load your app in the simulator, make some edits to your `main.lua` file in text editor, switch back to the simulator and relaunch your code to see the results. This makes it really easy to make edits and see the results, all while avoiding the time and trouble of quitting and restarting the simulator.

Basic Interactivity

Let's add some interactivity by creating a button that will change the color of the text randomly.

Starting with the "HelloWorld" project in the Sample Code, add the following lines at the end of `main.lua` to load an image:

```
local button = display.newImage( "button.png" )
button.x = display.stageWidth / 2
button.y = display.stageHeight - 50
```

This loads an image called `button.png` and positions it at the bottom center of the screen. It uses another `display` library function (`display.newImage`). This function returns an image object that we store in the variable `button`. We could have called the variable anything, but the name `button` seemed natural since we are going to turn this image into a button.

The image object that we created has built-in properties that we can modify. These include the `x,y` positions on the screen which refer to the position of the center of the image relative to the top-left corner of the screen.

To get a position towards the bottom of the screen, we took advantage of the `display` properties of the screen `display.stageWidth` and `display.stageHeight` to help us center the position of the image.

To turn the image into an actual button, we need to make it respond to events. There are various kinds of events. For this example, we will make the image respond to "tap" events (which are similar to single mouse clicks on a desktop computer). When you add the following lines at the end of `main.lua`, you can click on the image and the text color changes.

```
function button:tap( event )
    local r = math.random( 0, 255 )
    local g = math.random( 0, 255 )
    local b = math.random( 0, 255 )

    textObject:setTextColor( r, g, b )
end

button.addEventListener( "tap", button )
```

Let's see how this works. The code above is two parts. The first part defines an *object listener* for the image object `button`. An object listener (usually called a table listener) is an object method whose name matches the name of the event. Object listeners are just another name for object methods where a specific convention is followed: the name of the method is the same as the name of the event we are interested in, so in this case, we call the method `tap`. The colon is present because that's the syntax for defining object methods.

The second part is where we register this object listener to receive "tap" events. Fortunately, the image object `button` (like all objects created by the `display` library) has a built-in object method called `addEventListener` that allows us to make it interactive. Because it's an object method, the image object variable `button` is to the left of the colon and the object method `addEventListener` is to the right. The first argument is the name of the event and the second argument is the image object itself.

When the user taps on the image, the system sees that an object listener has been registered. It looks for an object method named `tap` inside that object and then calls that method. In our implementation of the `tap` object method, we generate 3 random numbers between 0 and 255 and use those to set the new text color.

The final code for this is available in the "HelloWorld2" Sample Code.

Animation and Sound

Let's animate the text and add some sound every time the user taps the button.

Start with the "HelloWorld2" project in the Sample Code and add the following lines at the end of `main.lua` so that the text will move vertically down by 100 pixels:

```
transition.to( textObject, { time=1000, y=textObject.y+100 } )
```

Here, we are using the `transition` library which does a lot of the heavy lifting on our behalf (see [Basic Animations](#)).

We can add some sound by adding one line to the `tap` object method:

```
function button:tap( event )
    local r = math.random( 0, 255 )
    local g = math.random( 0, 255 )
    local b = math.random( 0, 255 )

    textObject:setTextColor( r, g, b )
    media.playEventSound( "beep.caf" )
end

button.addEventListener( "tap", button )
```

Here we are using the `media` library which provides multimedia support.

Projects

Assets (Building Blocks)

An app consists of several building blocks also known as the *assets* of the application:

- **Source code:** all code is written in the Lua scripting language. Most of the standard Lua library APIs are available for your use. In addition there are APIs offered by AnscA that allow you to quickly and easily create graphically rich and interactive applications. Your main Lua application code is executed from a file called **main.lua**, but additional code may be placed in external Lua code files.
- **Non-code assets:** usually these are multimedia assets such as image (e.g. PNG or JPEG) files, sound files, and video files that are referenced in your application code. You are free to use your favorite creative tools to create these assets.

Starting a Project

Creating projects are really simple. All you do is create an empty folder in which to put all asset files used by your application. This includes any external library files that you wish to reference from your main application file, `main.lua`.

It is important that you only put files that your application actually uses. You should not put older versions of your `main.lua` file or older versions of your multimedia assets. Also, you should only place *final* production assets in this folder; you should *not* place the original native files produced by creative tools like Photoshop® (PSD) or Illustrator® (AI).

If you were to create a project called "MyProject", the directory structure would look like:

```
MyProject/  
  Icon.png  
  logo.png  
  main.lua  
  library.lua  
  library2.lua  
  photo.jpg  
  ...
```

Notice that the code and non-code assets (e.g. images) are in the **same** directory. There's also an icon image that lives alongside all the assets. This icon file should be a PNG that's 57x57 pixels.

For instructions on how to create your own external code libraries, see the **Module** section of **APIReference.pdf**. Additionally, for example code that loads optional Corona libraries, see the projects **Button** and **Movieclip** in the Sample Code directory of the Corona SDK.

Building for Device

See **DeviceBuildGuide.pdf** (in the Documentation folder of the Corona SDK) for instructions on how to build for your device or the iTunes App Store..

Sample Code

The SDK comes with sample code to help you get started:

GettingStarted

The following samples are no frills. They are bare bones to show you basic functionality:

- **Animation1** shows how to create a bouncing ball animation using `enterFrame` events.
- **Animation2** achieves same effect as *Animation1* using table listeners.
- **Animation3** shows 3 bouncing balls.
- **AnimationTime1** is a time-based version of *Animation1*.
- **EventSound** creates a basic metronome app.
- **FileDemo** shows how to create a new file and read an existing one.
- **FollowMe** shows how to make objects follow your finger.
- **HelloWorld** shows how to draw text and place an image on the screen.
- **HelloWorldLocalized** shows a translated “Hello World” depending on the language setting of your computer/device. If your native language is missing, send us a translation!
- **Orientation** shows how to make an app respond to orientation changes.
- **ReferencePoint1** shows how to rotate objects about an arbitrary reference point.
- **ReferencePoint2** animates the objects in *ReferencePoint1* as a collective group.
- **SimpleNetworkDownload** shows how to download an image from the internet.
- **Timer** shows how to achieve periodic calls.
- **Transition1** demonstrates how to do a simple fade out using the `transition` library.
- **Transition2** demonstrates how to sequence multiple transitions.

Interface

- **ActivityIndicator** shows how to make a spinning activity indicator appear.
- **Alert** shows how to generate a native iPhone alert dialog.
- **ButtonEvents** shows various buttons and how easy it is to create them using an external library.

Device

The following samples are best viewed on the device:

- **Camera** demonstrates capturing the contents of the screen to a file.
- **CaptureToFile** demonstrates capturing the contents of the screen to a file.
- **Compass** shows how to use the magnetometer hardware of the device.
- **GPS** shows how to use the GPS capabilities of the device.
- **StatusBar** shows how to change the status bar style.
- **WebOverlay** shows how to use the web popup feature to display HTML as a transparent overlay on top of animated Corona objects.

Graphics

- **Clock** shows a simple clock application.
- **Fishies** is a simple aquarium application.
- **Movieclip** shows how to create animated sprites using an external library.

Social

- **Twitter** shows how to login to your Twitter account and post a tweet.

Tutorial

- **HelloWorld**, **HelloWorld2**, and **HelloWorld3** are used in the [Tutorial Introduction](#).
- More coming!

Application Environment

Corona automatically adds key infrastructure into your application. This infrastructure handles user input and displays content on the screen for you, but you are responsible for configuring how user input is handled and what content goes on the screen to create the application's user interface, behavior, and features.

This chapter will give you an overview of the application-level architecture.

Life Cycle

When your application launches, some initialization work is performed on your behalf. After that, you have the opportunity to do initial setup such as defining functions, registering for events, drawing images, etc. via the code in `main.lua`.

Once all setup is complete, the application enters an event/drawing loop in which events trigger [listeners](#) in your code resulting in changes to the screen. See [Drawing Cycle](#) and [Screen Updates](#).

Keep in mind that the screen will *not* update until your initial setup is complete. Therefore, we recommend that you lazily perform operations. See [Minimize setup code](#).

Global Runtime Object

There is a global object called the `Runtime` object. This object's principal job is to allow you to register for events that have no specific target on screen such as "enterFrame" or "system" events (see [Event Basics](#)).

Sandbox

For security reasons, your application runs in its own sandbox. That means, your application has limited access to files, memory, network resources, etc. Practically speaking, your files (e.g. application images, data, preferences) are stored in a location that no other application can access. The paths to these files are unique to your application. Corona provides you with API's to generate these paths (see [Getting Paths to Files](#)).

Application Events

Termination

When the user hits the Home button, they are quitting your application. By registering for the `applicationExit` event (see [Registering for Events](#)), you have the opportunity to save any

unsaved data, save the state of the application, or perform cleanup such as deleting temporary files.

Interruptions

Your application can be interrupted by a variety of events. For example, your app may be interrupted by a phone call, an SMS message, a calendar alert, or the device going to sleep. Depending on the situation, the interruption may be temporary or may result in the termination of your application. Temporary interruptions may affect the timing of your application such as animations (see [Lost or Missing Time](#)). To handle these situations, you should register for the `applicationSuspend` and `applicationResume` events.

Customization

Application Icon

The application icon should be a 57 x 57 PNG image file. It should have the name `Icon.png` and be located in the `assets` project folder.

```
MyProject/  
  Icon.png <---  
  main.lua  
  ...
```

Note: the iPhone App Store requires a 512 x 512 pixel version of the icon so you should always create the icon in this higher-resolution.

Launch Image

When your application launches, you can choose to display a launch image before your application finishes initializing and is ready to display its interface. By using an image that looks like the initial user interface, you can create the illusion of a faster application launch. Alternatively, you can use this image for a “splash screen” displaying your application title or company logo.

The launch image should be named `Default.png` and be the dimensions of the screen. It should be located in the `assets` project folder.

```
MyProject/  
  Default.png <---  
  Icon.png  
  main.lua  
  ...
```

Event Handling

Events are the principal way in which you create interactive applications. They are a way of triggering responses in your program. For example, you can turn any display object into an interactive button. This flexibility is of the most unique things about the Corona SDK.

Global Events

Some events are broadcast, such as "enterFrame", "system", "orientation", etc. These events are global in nature because they are not directed at any particular object. Rather, they are broadcast to all interested listeners. The following is from the "Orientation" sample code. It demonstrates how your app can respond to orientation changes:

```
local label = display.newText( "portrait", 0, 0, nil, 30 )
label:setTextColor( 255,255,255 )
label.x = display.stageWidth/2; label.y = display.stageHeight/2

local function onOrientationChange( event )
    label.text = event.type -- change text to reflect current orientation
    -- rotate text so it remains upright
    local newAngle = label.rotation - event.delta
    transition.to( label, { time=150, rotation=newAngle } )
end

Runtime:addEventListener( "orientation", onOrientationChange )
```

Local Events

Local events are sent to a single listener and are not broadcast.

Hit Events

When the user's finger touches the screen, a hit event is generated and dispatched to display objects in the display hierarchy. By default, only those objects that intersect the hit location (the location of the finger on the screen) will be dispatched the event.

Propagation and Handling of Events

The events propagate through these objects in a particular order. By default, the first object in the display hierarchy to receive the event is the top-most display object that intersects the hit location; the next object is the next top-most object intersecting the hit location; and so on.

Hit events propagate until they are *handled*. You can stop propagation to the next *object* (all listeners of the current object still get the event) by telling the system that the event was handled. This boils down to making a listener return **true**. If at least one of the listeners of the current object returns **true**, event propagation ends; the next object will not get the event. If the event is still unhandled at the end of this traversal, it is broadcast as a global event to the global `Runtime` object.

Hit events are kind of a hybrid of local and global events. They are dispatched to a *single* display object at a time, but *any* listener of that object will be dispatched the event if it registered to receive that event.

Overriding Propagation with Focus

You can redirect future hit events to go to a specific display object by setting the *focus*.

Consider the situation of a rollover button. When a user presses on a button, the button should change its appearance in some way to indicate that the user is touching the button. If the user initially presses on the button and (without lifting) moves the finger off the button, the button should change to its original appearance.

This is very difficult to achieve using the default dispatch behavior and propagation rules of hit events. When a display object representing a rollover button is initially “hit”, we would like future events to go to it until the user lifts their finger off the screen. The way to achieve this is to set the *focus* on the display object. This instructs the system to deliver all future hit events to that display object:

```
function button:touch( event )
  local phase = event.phase
  if "began" == phase then
    -- Subsequent touch events will target button even if they are
    -- outside the stageBounds of button
    display.getCurrentStage():setFocus( self )
  else
    ...
  end

  return true
end
```

See the “Button” sample code for a complete example.

Listeners and Event Delivery

Listener can be either functions or a table (objects). In either case an event argument is always passed to the listener. Each kind of event stores different properties available for use.

Function Listener	Table Listener
<pre> local function listener(event) print("Call #"..event.count) end timer.performWithDelay(1000, listener, 5) </pre>	<pre> local listener = {} function listener:timer(event) print("Call #"..event.count) end timer.performWithDelay(1000, listener, 5) </pre>

Registering for Events

Events are registered with the target using the `addEventListener` *object* method. You pass the string name of the event you want to be notified of and the listener (function or table) that should handle that event. Often, the listener will be the same as the object as in the examples shown in [Basic Interactivity](#).

Conventions

All events have a `name` property that corresponds to the name you use to register the event.

Graphics and Drawing

All drawing that occurs on the screen is accomplished by creating `DisplayObjects`. Anything that appears on the screen is an instance of a `DisplayObject`.

Creating Display Objects

You don't actually create these objects directly. Instead, you create special kinds of `DisplayObjects` such as rectangles, circles, images, text, groups, etc.

These objects are all first-class citizens. You can change their position, rotate them, animate them, turn them into buttons, etc.

All of these objects share common properties and methods that are described in the [Display Objects](#) chapter of *Corona SDK Language and API Reference*.

All instances of `DisplayObject` can be treated like normal Lua tables. This means you can add your own properties to the object as long as they don't conflict with the names of `DisplayObject`'s predefined properties and method. The one exception is that you cannot index into a `DisplayObject` as an array using numerical indices.

Painter's Model

`DisplayObjects` are drawn to the screen using the Painter's Model of drawing. The easiest way to think of this is to imagine an actual painting. Here, the paint you apply at the beginning is below the paint you apply later. Each successive brush stroke obscures the strokes that came before.

You can think of a `DisplayObject` as analogous to a brush stroke. When you create a `DisplayObject`, you are "painting" a new object over existing display objects. As you draw more objects to the screen, the objects you draw last will obscure the ones you drew before.

Display Hierarchy

To manage the order in which `DisplayObjects` are drawn, `DisplayObjects` are organized in a hierarchy. This hierarchy determines which objects appear above other objects.

Group Objects

The hierarchy is made possible by the existence of group objects. Group objects are a special kind of `DisplayObject` that can have children. Group objects make it possible to organize your drawing so that you can build relationships between objects.

You can make any `DisplayObject` a child of a group. The children are organized in an array, so the first child (index 1) is below the next child, and so on; the last child is always above all its siblings. You insert objects into a group using the `group:insert()` object method and you access the children by indexing into the group with integer indices (e.g. `group[1]`):

```
local square = display.newRect( 0, 0, 100, 100 )
local rect = display.newRect( 0, 0, 100, 100 )
local group = display.newGroup()
group:insert( square )
group:insert( rect )
assert( (group[1] == square) and (group[2] == rect) )
```

Stage Objects

Whenever you create a new object, it is implicitly added to a special group object that is at the top of the hierarchy. This group object is called the *stage object* (note: we may rename this to screen object). Every time you create a `DisplayObject`, it is implicitly added to the stage object. By default, it will add that object at the end of the child array and thus appear above all other child display objects.

Moving Objects Forward and Backward

Unlike in real painting, the ordering of display objects is not set in stone; you can change the relative ordering of objects. The order in which a group object's children are drawn is determined by the ordering of the children array. Using the `group:insert()` object method, you can reorder the position of an object within its parent group. Conceptually, you can think of it as reinserting the object into the same parent group:

```
local square = display.newRect( 0, 0, 100, 100 ) -- Red square is
square:setFillColor( 255, 0, 0 )                -- at the bottom.
local circle = display.newCircle( 80, 120, 50 )  -- Green circle is
circle:setFillColor( 0, 255, 0 )                 -- in the middle.
local rect = display.newRect( 0, 0, 100, 100 )   -- Blue rect is
rect:setFillColor( 0, 0, 255 )                  -- at the top.

-- square,circle,rect all have same parent
local parent = square.parent

-- Move to top. Siblings at higher indices are above those at lower ones
parent:insert( square ) -- same as parent:insert( parent.length, square)

-- Move below all other siblings
parent:insert( 1, circle )
```


Drawing Cycle

The basic drawing model involves a cycle between executing Lua code and rendering objects in the display tree of the current stage object. During this cycle, the screen is only updated when objects in the display tree have changed. These changes occur by adding, removing, or changing properties of the child `DisplayObjects`.

Currently, this cycle occurs 30 times a second. At the beginning of each cycle, an "enterFrame" event is dispatched to any registered listeners in your Lua code. Once all listeners have *finished* executing, the screen is updated.

Screen Updates

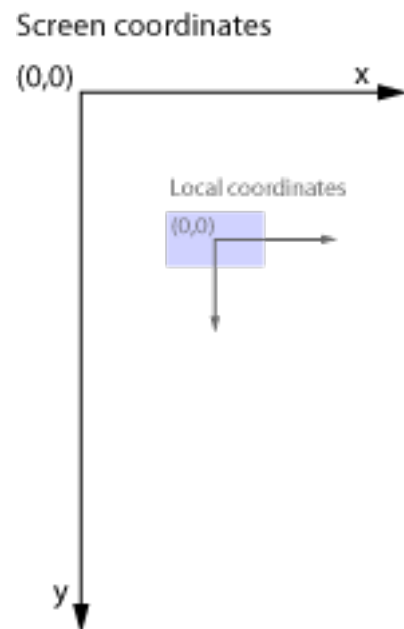
The screen never updates while a block of your Lua code is executing. Therefore, if you modify a display object multiple times in a code block (e.g. the x position property), only the last change (e.g. the final value of x) will be reflected by the screen update.

Coordinates and Transforms

Coordinate spaces define the location in which all drawing occurs. The screen represents the base coordinate system for drawing. All content must eventually be specified relative to the origin of the screen.

Often, it is unwieldy to describe everything in terms of screen coordinates. Therefore, we introduce the concept of local coordinates. Every display object operates in their own local coordinate system. The heavy lifting of converting between a display object's local coordinates and the screen coordinates is done for you.

The process of translating between local coordinates and screen coordinates is made possible by mathematical transforms, or transforms, for short. Transforms convert coordinates from one space to another.



Coordinates

A Cartesian coordinate system (also known as a rectangular coordinate system) is used to define position. Unlike standard Cartesian coordinates, the origin of the screen is located at the top-left corner so that positive y-coordinate values extend downward (positive x-values extend to the right as usual). All screen coordinates are defined relative to this origin.

Local coordinates allow you to manipulate geometric properties of a display object such as rotation in a more intuitive fashion. Every display object has an origin relative to its parent's. This origin essentially defines the position of the display object (relative to its parent). For

example, if variable `r` was a rectangle, then its origin/local-position would be (`r.xOrigin`, `r.yOrigin`).

Every object also has a local reference (or registration) point about which transformations such as rotations occur. The reference point is defined by two numbers (in the case of the rectangle `r`, these are `r.xReference`, `r.yReference`) which specify the location of the reference point relative to the local origin. By default, the reference point is the same as the local origin, i.e. the two numbers of the reference point are (0, 0).

Changing Position of Objects

To change the position of a display object, you can either manipulate the (`xOrigin`, `yOrigin`) properties or the (`x`, `y`) properties — typically you do the latter because that's the same point about which scales and rotations occur.

The (`xReference`, `yReference`) properties do not affect the position of a display object. Rather, they define **where** the reference point will be located, as this location affects scaling and rotation.

Transforms

Often, keeping track of transformations is a very error-prone and tricky business. This is because the order in which geometric transformations occurs determines the final position. For example, rotating an object and then scaling it (non-uniformly) will generate a different result from scaling that object first and then rotating.

To simplify things, we define an order of operations when transforming the object. These operations are all relative to the reference point of the display object. In this way, you are free to change the value of an object's position, rotation, and scale properties in any order you please; the resulting transformation remains consistent.

This transformation is calculated by applying geometric operations in the following order:

1. Scale the display object about its reference point using (`object.xScale`, `object.yScale`).
2. Rotate about the display object's reference point `object.rotation` degrees.
3. Move the object's origin (not its reference point) relative to the parent's by (`object.x`, `object.y`) in local coordinates.

Note: the methods `object.scale()`, `object.rotate()`, and `object.translate()` merely change the value of the underlying geometric properties. The order in which you call them does not affect the final result, so you should **not** think of them as matrix transformations.

Object References

Because objects can be reordered in the hierarchy, using integer indices to access children of groups is fragile. If you move a child above its sibling, all integer indices have to be updated. A simple solution to this is to store child display objects as a property of the parent group. This makes it easier to access those objects later.

Let's consider a situation where we have images for the sun and the planets of our solar system. We want to put them all under one group. In this example, we have a table listing all the files and we've created a group that we'll store the image objects in.

```
local planetFiles = { sun="sun.png", mercury="mercury.png",
  venus="venus.png", earth="earth.png", mars="mars.png",
  jupiter="jupiter.png", saturn="saturn.png", neptune="neptune.png",
  uranus="uranus.png", pluto="pluto.png" }

local solarSystem = display.newGroup()
```

The next step is to create the image objects by iterating through the table `planetFiles`. We use a special iterator `ipairs` which will return both the property name and the image filename stored in the property of `planetFiles`. We use the filename to load the image; we use the property name to assign a property in `group` so we can easily refer to it later in the group without worrying about integer indices:

```
-- Loop through all the files, load the image, assign property in the group
for key,file in pairs( planetFiles ) do
  -- key will be "sun", "mercury", etc.
  -- file will be "sun.png", "mercury.png", etc.
  local planet = display.newImage( file )
  solarSystem:insert( planet )
  solarSystem[key] = planet
end

-- Afterwards:
-- solarSystem.sun will refer to the image object for "sun.png",
-- solarSystem.mercury will refer to the image object "mercury.png",
-- etc.
```

Keep in mind that if you want to remove one of these objects, you need to do two things. First, you need to remove it from the display hierarchy. Second, you need to set the corresponding property of the parent group to **nil** (see [Variable References](#)).

Removing Objects Properly

Because devices have limited resources, it is important to remove display objects from the display hierarchy when you no longer use them. This helps overall system performance by reducing memory consumption (especially images) and eliminates unnecessary drawing.

When you create a display object, it is by default added to the root object of the display hierarchy. This object is a special kind of group object known as the stage object.

To properly remove an object so it no longer renders on screen, you need to remove the object explicitly from its parent. This removes the object from the display hierarchy:

```
image.parent:remove( image ) -- remove image from hierarchy
```

However, this is not always sufficient to free the memory consumed by the image. To ensure that the image object is garbage collected properly, we need to eliminate all variable references to it as we will explain in the next section.

Variable References

Even though a display object has been removed from the hierarchy, there are situations in which the object continues to exist. In our above example, the parent group `solarSystem` stores references to the image objects for planets as properties. So even after removing an image from the display hierarchy, we still need to ensure that `solarSystem` no longer refers to the image. To do this, we set the property to **nil** (we call this **nil**'ing out the property).

```
local sun = solarSystem.sun
sun.parent:remove( sun ) -- remove image from hierarchy
solarSystem.sun = nil -- remove sun as a property of solarSystem
```

Generally speaking, if you inserted the display object as a table element (e.g. as a property of the table or as an array element), the display object will remain in existence even though it does not display to screen (see [Object References](#)). You have to **nil** out the property as in the above example.

Similarly, if other variables that point to the display object, the display object cannot be freed as long as those objects continue to exist. For example, global variables are never freed so if a global variable points to a display object, it will continue to exist even if it is not in the display hierarchy. Here, you should also set the global variable to **nil** when you no longer need it.

Another subtlety is when a function refers to a local variable outside its scope:

```
local sun = solarSystem.sun

function dimSun()
    sun.alpha = 0.5 -- sun was declared outside the function block
end
```

In this case, there is still an outstanding reference to the image object inside this function. Because this function is global, the image object must remain in existence. There are 2 solutions: make the function non-global (i.e. local) or change the function so that no variables outside the function block are referenced. The latter is preferable and is also more general in that it can be applied to any display object:

```
local sun = solarSystem.sun

function dim( object )
  object.alpha = 0.5
end
```

Common Pitfalls

A common mistake is to improperly remove all objects from a group. This typically happens when you write code that iterates through a group attempting to remove each child from the display hierarchy. It's natural to iterate through the group in the *forward* direction. However, this can cause no end of confusion.

Continuing with our solar system example, consider the following where we attempt (incorrectly) to remove all the objects from the solar system.

```
for i=1,solarSystem.numChildren do
  local child = solarSystem[i]
  child.parent:remove( child )
end
```

The problem here is that we are modifying a collection (i.e. the group's children array) as we iterate through that same collection. The result is we remove every other child. The easiest way to illustrate this is with a parallel example involving an array of integers:

```
local array = {1,2,3,4,5,6,7,8,9,10}
print( table.concat( array, " " ) ) --> 1 2 3 4 5 6 7 8 9 10

for i=1,#array do
  table.remove( array, i )
end

print( table.concat( array, " " ) ) --> 2 4 6 8 10
```

The fix is to iterate *backwards*.

```
for i=solarSystem.numChildren,1,-1 do
  local child = solarSystem[i]
  child.parent:remove( child )
end
```

Of course, this only ensures that all children have been removed from the display hierarchy; you still have to set all references to these display objects to **nil**. So in this example, we were merely trying to illustrate the highlight the pitfalls of iterating forward through the children of

a group and modifying the group at the same time. A good implementation would also set the corresponding properties in `solarSystem` to **nil** for proper cleanup.

Animation

One of the most powerful things about the Corona SDK is that any display object can be animated. This is a testament to the flexible graphics model that Corona offers.

Animations allow you to create visually-rich and engaging user experiences. Animations are accomplished by generating a sequence of frames that evolve smoothly from frame to frame. The term *tween* (short for inbetween) is a term describing the process in which such intermediate frames are generated. It is often used as shorthand to indicate that a property of an object will change during the animation, as in tweening the position.

Basic Animations

The `transition` library allows you to easily create animations with only a single line of code by allowing you to tween any property of a display object. For example, you can fadeout a display object by tweening its alpha property (the alpha property transitions from 1.0 to 0).

The simplest way to do this is to use the `transition.to` method which takes a display object as its first argument and a table containing the control parameters as its second. The control parameters specify the duration of the animation, an optional delay for when to start the animation, and the final values of properties for the display object. The intermediate values for a property are determined by an easing function that is also specified as a control parameter. By default this is a linear function.

Below are some examples of how to animate a square (see `Transition2` in the sample code):

```
local square = display.newRect( 0, 0, 100, 100 )
square:setFillColor( 255,255,255 )

local w,h = display.stageWidth, display.stageHeight

local square = display.newRect( 0, 0, 100, 100 )
square:setFillColor( 255,255,255 )

local w,h = display.stageWidth, display.stageHeight

-- (1) move square to bottom right corner; subtract half side-length
--     b/c the local origin is at the square's center; fade out square
transition.to( square, { time=1500, alpha=0, x=(w-50), y=(h-50) } )

-- (2) fade square back in after 2.5 seconds
transition.to( square, { time=500, delay=2500, alpha=1.0 } )
```

In the first tween, notice that multiple values are changing: position and alpha. That's because in the control parameters we specified the final values for the `x`, `y`, and `alpha` properties. For each property specified in the control parameters, the library looks at the current property value and gradually changes that property to the final value over the time period specified (1.5

seconds in this case). In the last tween, we use the `delay` control parameter to start the tween after the initial tween's fadeout is complete.

Note that the `transition` library operates in a [time-based](#) manner.

Animated Sprites (or "Movieclips")

The external `sprite` library allows you to create animated sprites (sometimes called "movieclips") from sequences of images, which can then be moved around the screen using exactly the same techniques as the simpler images discussed here. Functions are available to play these animations in either the forward or reverse direction, and to jump to specified frames within the sequence.

For more information on animated sprites, see the **sprite** section of the **API Reference**. For a sample project using the `sprite` library, see the **Movieclip** project in the Sample Code directory of the Corona SDK.

Custom/Programmatic Animations

Often you will need to create your own custom animations that are not feasible using the `transition` library. This is known as programmatic animation because you have to write custom code to produce the animation sequence.

To create such animations, you need to change the contents of the screen over time. In some environments, it's natural to do this by changing properties of an object in loops such as a **for** or **while** loop. However, in Corona, you cannot produce animations using such loops because the screen is **never** updated within a block of code (see [Screen Updates](#)).

Instead, animations are produced by repeatedly calling listeners. These listeners modify the display objects on the screen and then exit, thus allowing the screen to be updated. Such listeners are known as "enterFrame" listeners because you register these listeners with the "enterFrame" event.

In the drawing cycle, "enterFrame" events are dispatched before the screen is updated providing your code the opportunity to modify the contents of the screen (see [Drawing Cycle](#)). With "enterFrame" events, you can produce all kinds of animations. In fact, the `transition` library is built on top of these events.

Below is an example of how to animate a bouncing ball:


```

local xdirection,ydirection = 1,1
local xpos,ypos = display.stageWidth*0.5,display.stageHeight*0.5
local circle = display.newCircle( xpos, ypos, 20 );
circle:setFillColor(255,0,0,255);

local function animate(event)
    xpos = xpos + ( 2.8 * xdirection );
    ypos = ypos + ( 2.2 * ydirection );

    if (xpos > display.stageWidth - 20 or xpos < 20) then
        xdirection = xdirection * -1;
    end
    if (ypos > display.stageHeight - 20 or ypos < 20) then
        ydirection = ydirection * -1;
    end

    circle:translate( xpos - circle.x, ypos - circle.y)
end

Runtime:addEventListener( "enterFrame", animate );

```

The listener function `animate` is called every time an `"enterFrame"` event occurs. It is responsible for changing the position of the ball and for ensuring that the ball “bounces” when it hits the edge of the screen.

Because `"enterFrame"` events occur at the global level, you register listeners for those events with the global `Runtime` object.

Frame Rate

The `"enterFrame"` event occurs at a regular interval known as the frame rate, so your listeners will be called at the frame rate. However, if your listeners take too long to exit, then the *actual* frame rate will be less than the desired frame rate.

Time-based vs Frame-based

In the above example, the animation was done in a frame-based manner. If the actual frame rate were to slow down, the ball would appear to move more slowly as each and every intermediate frame got rendered; no intermediate frames would be skipped. If you were trying to synchronize the animation with sound, then this behavior would be extremely problematic.

The solution is time-based animation. We can transform the above example to be time-based by calculating how much time had passed between calls to our listener and changing the velocities appropriately. This would result in the following changes:

```

local xdirection,ydirection = 1,1
local xpos,ypos = display.stageWidth*0.5,display.stageHeight*0.5
local circle = display.newCircle( xpos, ypos, 20 );
circle:setFillColor(255,0,0,255);

local tPrevious = system.getTimer()
local function animate(event)
    local tDelta = event.time - tPrevious
    tPrevious = event.time
    xpos = xpos + ( 0.084*xdirection*tDelta );
    ypos = ypos + ( 0.066*ydirection*tDelta );

    if (xpos > display.stageWidth - 20 or xpos < 20) then
        xdirection = xdirection * -1;
    end
    if (ypos > display.stageHeight - 20 or ypos < 20) then
        ydirection = ydirection * -1;
    end

    circle:translate( xpos - circle.x, ypos - circle.y)
end

Runtime:addEventListener( "enterFrame", animate );

```

Notice how we leverage the fact that the "enterFrame" event contains a property storing the time in milliseconds. We compare that with the previous time to determine how far the ball should travel. In addition, our old x,y velocities (2.8, 2.2) implicitly assumed that time was measured in frames. The equivalent time in milliseconds is simply the frame rate. By default, that's set to 30 fps or 33.3 milliseconds. So we can multiply the old velocities by (30/1000) to get the new time-based velocities.

Lost or Missing Time

The one problem to watch out for when doing time-based animation is that when the device suspends the app, you need to account for the "lost" time. In the simulator, you can simulate a suspend by using the keyboard shortcut ⌘↓ (command-down arrow) which corresponds to the **Suspend/Resume** menu item under **Hardware**. (Note a known issue is that clicking on the menu causes the app to pause as if it were suspended but doesn't generate a suspend event).

In the bouncing ball animation, if the app is suspended for half a second, the ball may appear to jump across the screen. In the example above, the solution is to adjust `tPrevious` to account for this missing time:

```
-- Add the following below the code in the previous example

local tSuspend
local function onSuspendResume( event )
  if "applicationSuspend" == event.type then
    tSuspend = system.getTimer()
  elseif "applicationResume" == event.type then
    -- add missing time to tPrevious
    tPrevious = tPrevious + ( system.getTimer() - tSuspend )
  end
end

Runtime:addEventListener( "system", onSuspendResume );
```

User Interface

Buttons

The *external ui* library allows you to easily create buttons with normal and rollover states, by assigning an image file for each state. These buttons can optionally call custom functions on both the “press” and “release” event.

For more information on creating buttons, see the **ui** section of the **API Reference**. For sample projects using the *external ui* library, see the **Button** and **Movieclip** projects in the Sample Code directory of the Corona SDK.

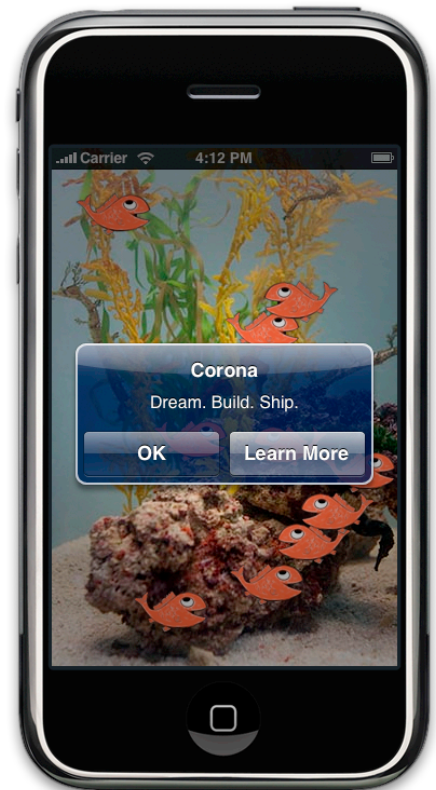
Alerts

The built-in *native* library allows you to launch native iPhone alert dialogs with one or more buttons, and to assign functions to the different buttons.

For more information on native alerts, see the **native** section of the **API Reference**. For a sample project using the *native* library, see the **Alert** project in the Sample Code directory of the Corona SDK.

Text Input

(Forthcoming.)



Files

Applications are sandboxed (see [Sandbox](#)) meaning your files (e.g. application images, data, preferences) are stored in a location that no other application can access. Your files will reside in an app-specific directory for documents, resources, or temporary files.

Getting Paths to Files

The paths to these files are unique to your application. To create file paths, you use the `system.pathForFile` function. The following generates an *absolute path* to the icon file for your application using the application's resource directory as the base directory for "Icon.png":

```
local path = system.pathForFile( "Icon.png", system.ResourceDirectory )
```

In general, your files must reside in one of 3 possible base directories:

- `system.DocumentsDirectory` should be used for files that need to persist between application sessions.
- `system.TemporaryDirectory` is a temporary directory. Files written to this directory are not guaranteed to exist in subsequent application sessions. They may or may not exist.
- `system.ResourceDirectory` is the directory where all application assets exist. Note: you should **never** create, modify, or add files to this directory (see [Beware Security Violations](#)).

Reading Files

To read files, you use the `io` library provided by Lua. That library allows you to open files given an absolute path. Here's an example of how to open a file and read all its contents.

```
local path = system.pathForFile( "data.txt", system.DocumentsDirectory )
local file = io.open( path, "r" )
if file then -- nil if no file found
    local contents = file:read( "*" )
    print( "Contents of " .. path .. "\n" .. contents )
    io.close( file )
end
```

In this example, we assume that a file "data.txt" exists in the documents directory. The function `io.open` creates a file object in which several object methods are available. We use the object method `read` to obtain the contents of the file as a string. At the end, we tell Lua that we are

done with the file using the function `io.close` to close the file. After that point, the variable `file` is no longer a valid file object.

Writing Files

To write files, you follow many of the same steps as reading a file. Instead of using a read method, you write data (strings or numbers) to a file. Here's an example that builds on the previous. It looks for the file "data.txt" and reads it as before; however if it does not exist, then a file is created. Note the addition of hard line breaks (paragraph breaks) using the special `\n` character.

```
local path = system.pathForFile( "data.txt", system.DocumentsDirectory )

-- io.open opens a file at path. returns nil if no file found
local file = io.open( path, "r" )
if file then
    -- read all contents of file into a string
    local contents = file:read( "*" )
    print( "Contents of " .. path .. "\n" .. contents )
    io.close( file )
else
    -- create file b/c it doesn't exist yet
    file = io.open( path, "w" )
    local numbers = {1,2,3,4,5,6,7,8,9}
    file:write( "Feed me data!\n", numbers[1], numbers[2], "\n" )
    for _,v in ipairs( numbers ) do file:write( v, " " ) end
    file:write( "\nNo more data\n" )
    io.close( file )
end
```

Beware Security Violations

Every time your application is launched, the integrity of the application is verified. It must be the same as the original file that was installed. If the integrity check fails, your app will not launch. Therefore, you should **never** create, modify, or write to files in the resources directory (see [system.ResourceDirectory](#)).

Networking

Corona includes the latest version (v2.02) of the **LuaSocket** libraries. These Lua modules implement common network protocols such as SMTP (sending e-mails), HTTP (WWW access) and FTP (uploading and downloading files). Also included are features to support MIME (common encodings), URL manipulation and LTN12 (transferring and filtering data).

Full LuaSocket documentation can be found here:

<http://www.tecgraf.puc-rio.br/~diego/professional/luasocket/reference.html>

Although these libraries are built into Corona, you need to use the Lua `require` syntax to make the functions available to your code. See the **Module** section of the **API Reference** for more information on how to use `require`.

For more information on network syntax, see the **Network** section of the **API Reference**.

Downloading Files

For a sample project that demonstrates remote image download over the network, see the **SimpleImageDownload** project in the Sample Code directory of the Corona SDK.

Uploading Files

Multimedia

Playing Sound

Event Sound

If you have short sounds that are roughly 1-3 seconds in duration, you should use the event sound API's. The following is an example of a metronome that plays a beep every second.

```
local soundID = media.newEventSound( "beep.caf" )
media.playEventSound( soundID )
local playBeep = function()
    media.playEventSound( soundID )
end
timer.performWithDelay( 1000, playBeep, 0 )
```

The above example demonstrates best practices when you play the same event sound repeatedly by only loading the event sound file once using `media.newEventSound` to load the sound file. The result of that function is a sound id that is passed as an argument to `media.playEventSound` in the `playBeep` function.

We could have just used `media.playEventSound` and always pass the sound file, but that would have been wasteful b/c every time `playBeep` was called, we would have to load the same sound file from disk.

Longer Sounds

For longer sounds, there are several functions that allow you to play, pause, and stop playing a sound. However, you can only have one such long sound file open at a time. These functions all operate on the sound that is opened by the previous call to `media.playSound`.

```
media.playSound( "song.mp3" )

local stopAfter10Seconds = function()
    media.playStopSound()
end
timer.performWithDelay( 10000, stopAfter10Seconds )
```

Playing Video

Video playback relies on a device-specific popup media player. During video playback, the media player interface takes over. An important thing to note is that `media.playVideo` is

asynchronous. Once the code block is exited, the application will be suspended until the video playback is complete.

Therefore, if you want to be notified of when the media player is exited (e.g. video finished playing or the user cancelled video playback), you should register a listener:

```
local onComplete = function(event)
  print( "video session ended" )
end
media.playVideo( "Movie.m4v", true, onComplete )
```

Device Support

Discussions are forthcoming. See *API Reference* for a description of API's available.

Orientation

See "orientation" events in the *API Reference*

Camera

See the `media` library in the *API Reference*

Common Design Tasks

Strategies to Avoid Globals

Overcoming Scoping Issues of Locals

Keeping Everything Local in Listeners

Pausing and Restarting Animations

If you have [programmatic animations](#), you can pause and restart them. Here's how you can make a button start and stop an animation:

```
local logo = display.newImage( "logo.png", 160, 240 )
function logo:enterFrame(event)
    -- do something like make the logo bounce around the edges of the screen
end

Runtime:addEventListener( "enterFrame", logo );

function logo:tap( event )
    if logo.isPaused then -- initially nil which is false anyways
        Runtime:removeEventListener( "enterFrame", self )
    else
        Runtime:addEventListener( "enterFrame", self )
    end
    return true -- we handled the event so don't propagate
end

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

Managing Screens

When you design your application interface, you'll likely find yourself storyboarding in terms of "screens" such as the splash screen, the home screen (sometimes called the main screen or menu screen), or some other screen.

Group objects are the perfect way to manage the content for each screen. In this way, you can create a group for the splash screen, for the home screen, and for any other screen. You can then leverage the `transition` library to create animated transitions (fades, sliding, etc) back and forth between screens.

Saving Data on Application Exit

When you save data to a file, you need to decide where you are going to put it, what the name of the file is, and what data you need to save.

Typically, you would put this in the documents directory of your application's sandbox.

```
local path = system.pathForFile( "data.txt", system.DocumentsDirectory )

-- io.open opens a file at path. returns nil if no file found
local file = io.open( path, "r" )
if file then
  -- read all contents of file into a string
  local contents = file:read( "*" )
  print( "Contents of " .. path .. "\n" .. contents )
  io.close( file )
else
  -- create file b/c it doesn't exist yet
  file = io.open( path, "w" )
  local numbers = {1,2,3,4,5,6,7,8,9}
  file:write( "Feed me data!\n", numbers[1], numbers[2], "\n" )
  for _,v in ipairs( numbers ) do file:write( v, " " ) end
  file:write( "\nNo more data\n" )
  io.close( file )
end
```

Restoring Data on Application Launch

In certain situations, it is desirable to resume where the user left off between launches. In order to accomplish this, you need to register for the appropriate system events. Here is a skeleton of how your app might be structured:

```
local function shouldResume()  
    -- return true or false depending on whether we need to resume  
end  
  
local function onSystemEvent( event )  
    if event.type == "applicationExit" then  
        -- save stuff to disk  
    elseif event.type == "applicationStart" then  
        if shouldResume()  
            -- load stuff off disk  
        else  
            -- start app up normally  
        end  
    end  
end  
  
Runtime:addEventListener( "enterFrame", onSystemEvent );
```

Performance and Optimization

As you develop your application, you should always consider how your design choices affect the performance of your application. Despite recent improvements in computing power, mobile devices still face fundamental constraints in processing power, memory usage, and battery life. Therefore, it's best to think of performance and optimization not only in achieving faster response times but also in minimizing memory usage **and** maximizing battery life.

Using Memory Efficiently

Memory is a critical resource on mobile devices. Some devices may even forcibly quit your application if you consume too much of it.

- **Eliminate memory leaks.** Your application should not have any memory leaks. Allowing leaks to exist means your application may not have the memory it needs later. Although Lua does automatic memory management, memory leaks can still occur in your code (see [Memory Allocation](#)). For example, global variables are never considered garbage; it is up to you to tell Lua that these variables are garbage by `nil`-ing them out (`globalVar = nil`). If a global variable is a table, then any items in that table will not be considered garbage until you `nil` them out (`globalVar.item = nil`).
- **Make resource files as small as possible.** Resource files used by your application typically reside on the disk. They must be loaded into memory before they can be used. Images should be made as small as possible. For example, it's often tempting to load multiple fullscreen-sized images to create flipbook-style animation, even though only elements in the foreground change. In such situations, it's much better to separate the background into a single fullscreen-sized images and animate the foreground using much smaller images.
- **Load resources lazily.** Avoid loading resource files until they are actually needed. Intuitively, prefetching resource files might seem like a good way to save time; however, this practice can actually backfire slowing down your application, because of the way a device responds to low-memory situations. In addition, your application may never even use the resource, making the prefetch a waste of time and memory.
- **Remove objects from the display hierarchy.** When a display object is created, it is implicitly added to a display hierarchy. When you no longer need a display object, you should remove it from the display hierarchy, especially then the objects contain images. This makes the display object eligible for garbage collection. However, this is no guarantee that the object will be considered garbage because other variables (not considered garbage) might be referencing it. In this case, the object will not render to the screen, but its memory isn't freed either.

Example

Below is an example of how a memory leak can occur. The code on the left removes the rectangle from the display hierarchy once you tap on it, but the memory used by the rectangle leaks because the variable `rect` that still refers to it. Because `rect` is a global variable, the

display object it references will not be freed even though the rectangle no longer renders on the screen.

One way to fix this is to modify the `removeOnTap` function adding a line to `nil`-out the reference (`rect = nil`). The problem with this approach is that you can no longer use this function for other objects. Instead, you'd have to duplicate the code for that function differing only in one line. A better solution is to simply make the global variable a local one, so you no longer have to explicitly `nil`-out the reference. The code on the right fixes this by turning `rect` into a local variable.

Bad (Memory Leak)	Better
<pre>-- rect is a global variable rect = display.newRect(0,0,10,10) rect:setFillColor(255,255,255) local function removeOnTap(event) local t = event.target local parent = t.parent -- remove from display hierarchy -- but var "rect" still exists -- so memory is never freed parent:remove(t) return true end rect:addEventListener("tap", removeOnTap)</pre>	<pre>-- rect is a local variable local rect = display.newRect(0,0,10,10) rect:setFillColor(255,255,255) local function removeOnTap(event) local t = event.target local parent = t.parent -- remove from display hierarchy parent:remove(t) return true end rect:addEventListener("tap", removeOnTap)</pre>

Reducing Power Consumption

Battery life is inherently limited on mobile devices because of their small form factor. You can help improve battery life by minimizing the use of the following features:

- Network traffic (Wi-Fi radios and baseband cell radios)
- GPS
- Accelerometers
- Disk accesses (reading/writing to files)

You will inevitably use these features to create great user experiences. However, as you design your application, make judicious use of these features, as everything you do impacts the battery life of the user's device

Network

Of all the activities, network accesses consume the most power. You can minimize the impact of network traffic by following these guidelines:

- **Do not poll.** Connect to external network servers only when needed.
- **Minimize data size.** Optimize the data you transmit so that it is as small as possible.
- **Transmit in bursts.** More power is consumed the longer the radio is actively transmitting data. Therefore, transmit the data in bursts instead of spreading out the same data into smaller transmission packets over time.
- If you access location data, stop collecting location update events as soon as you have the data you need. Location data comes from using GPS, cell, and Wi-Fi networks, so the best way to save power is to only collect location data when you need it.

CPU

Another way to minimize power consumption is to optimize the running time of your application. Some rules of thumb include performing work lazily on an as needed basis rather than doing work that ends up being unused.

Graphics

Group objects

If you are going to set the property (such as alpha) of a bunch of objects to the same value, it's preferable to add the objects to a group and then modify the property of the group. It's easier for you to code and it optimizes your animation.

Another benefit of organizing objects into groups is to organize your content into screens (see [Managing Screens](#)).

Turn off animations for non-visible objects

It may sound obvious, but it's often easy to overlook the fact that you may have animations running that are invisible or that are offscreen.

For example, you might have a group that stores all objects for the main menu. In the main menu group, there are several child objects that you animate (perhaps a bouncing ball or a rotating gear) by registering a listener for "enterFrame" events to achieve custom animation. When the user goes to another screen, you set the group to be invisible (`isVisible` set to **false**). Unfortunately, the listener will continue to do calculations that don't produce any visible effect. See [Pausing and Restarting Animations](#).

The solution is to remove the event listener when you change to a new screen and then re-register the listener when you re-enter the menu screen.

Optimize image sizes

Be careful when using large images, especially fullscreen images. They impact performance in two ways:

First, they take more time to load so they can impact the responsiveness of your application.

Second, they use up a lot of memory; some devices will even force quit your application if too much memory is consumed. Therefore, you should remove them from their parent group when you no longer need them:

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

-- do stuff with image

image:getParent():remove( image )
```

Minimize setup code at startup time

When your application launches, your `main.lua` file will typically contain a lot of setup code to add images to the screen, set up listeners to respond to user events or frame events, etc. If your setup code takes too long, your users will not see any screen updates because no screen update can occur until after a code block has finished executing.

Lua: Best Practices

Simple changes to your Lua code can yield tremendous benefits. Below are some examples of ways to squeeze out extra performance in your Lua code using very simple coding changes:

Use locals (i.e. avoid global variables)

Avoid global variables. Period. In Lua, you will sacrifice performance if you use global variables. When in doubt, precede your variable declarations with `local`.

This applies even to functions. In Lua functions are variables too. In long loops, it's better to assign a function to a local variable. In the following example, the code on the left runs **30% slower** than the one on the right!

Global	Local
<pre>for i = 1, 1000000 do local x = math.sin(i) end</pre>	<pre>local sin = math.sin for i = 1, 1000000 do local x = sin(i) end</pre>

In some situations, you may not be able to cache a global variable in a pure local variable. For example, inside a function, you may not be able to cache the function as a variable local to

that function. In this situation, you can use local variables that are outside the function scope, i.e. external locals. External locals are not as fast as pure locals, but are faster than globals.

In the following example, we can optimize the code on the left by declaring `sin` once outside function `foo`:

Global	External local
<pre>function foo (x) for i = 1, 1000000 do x = x + math.sin(i) end return x end</pre>	<pre>local sin = math.sin function foo (x) for i = 1, 1000000 do x = x + sin(i) end return x end</pre>

Again, the global version runs 30% slower.

Math: fast vs slow

Multiplication $x*0.5$ is faster than division $x/2$.

$x*x$ is faster than x^2

Inserting objects into arrays

Short inline expressions can be faster than function calls. For example, when appending `item` to an array `t` the following `t[#t+1] = item` is much faster than `table.insert(t, item)`.

Constant Folding

Constant folding is the process of simplifying constant expressions at compile time. For example, the statement `i=111+111` will be just as fast as `i=222` because the compiler can precalculate the value of that expression.

To take advantage of this, you need to be aware of when the compiler can do such calculations and when it cannot. First, the compiler is not smart enough to know that values inside variables are constant even if you consider those variables to be constant in your code.

Second, because of associativity rules, the ordering of constants matters. Lua considers `a+b+c` to be equivalent to `(a+b)+c`. Therefore, the expression `1+2+x` will be treated as `(1+2)+x` and be optimized to `3+x`, but `x+1+2` will be treated as `(x+1)+2` and thus not be optimized.

Cache properties in a local variable

If you are constantly accessing a property of a table but not changing its value, then you should cache that value. There is a slight performance penalty to doing property lookups in a table. For objects created by Anscas's api's — such as the display object returned by `display.newImage()` — the penalty is even higher.

Uncached	Cached
<pre>function foo (o) local n = 0 for i = 1, 1000000 do -- lookup o.x each time n = i + n * o.x end return n end</pre>	<pre>function foo (o) -- cache o.x before the loop local x = o.x local n = 0 for i = 1, 1000000 do n = i + n * x end return n end</pre>

Tuning Your Code

Revision History

This table describes the changes to *Applications Programming Guide*:

Date	Notes
2009-06-15	Initial draft
2009-11-28	Revised for product release 1.0