Inga Pflaumer

Making games in Corona using Lua

Developed for Women Who Code Sydney

Intro

Before starting to work on this workshop, you need to have Corona SDK downloaded and installed. We won't spent time on the installation process as it is relatively straightforward, but if you have any issues, please refer to Corona Developer Guide.

The goal of this workshop is to show you that making games can be easy, fun and hopefully exciting.

This is not, in any way, a workshop about best programming practices or high level concepts. We won't touch many of really important concepts and structures, my goal is for each and every one of you to walk away with a working prototype. For now - we are making a small game using a new programming language and framework, and we are doing it fast.

So what do you need to create a game? Generally you need some story, art, music and code. And I do this all on my own, but the internet is a magical place – so you can find lots of free stuff under Creative Commons licence. All the art and music we are going to use for today's project was found online, and is free to use.

When you found or created a story, art and music that you like, the last step is to put it all together into a game. And that is what we are doing now.

We are going to make a simple dancing game. The game will display arrows with directions on the screen (I'll be calling them "instructions"), and when player swiped in the right direction - their character moves into a dancing position.

Part 1. Creating a new project in Corona

Open Corona Simulator and click on “open project”, then find the folder with workshop files.

For this workshop you’ll need to have several programs opened at the same time.

1. Corona Simulator
2. Corona Simulator console (it opens when you open simulator)
3. Your project folder in Finder or My Computer.
4. Your text editor.
5. This workshop, and additional files from workshop folder

In your project folder you’ll see a number of icons and several files. The main files you need are main.lua and main_part1_ready.lua.

Switch to **Corona Simulator**. And look at menu “Window” -> “View as”

You’ll see a number of devices available in the simulator. We’ll develop this project for iPhone 5. But the process is pretty much the same for all devices, and we’ll use some features like resizing and positioning to make sure that when you changed the device, your game still looks pretty.

Now let’s look at our Corona Simulator console. It is a really helpful tool that we’ll use to debug our code. When you have a code error - it will show you what’s wrong and where to look for it.

Also we’ll use function print() to display some stats – for instance, when we need to see the counter for our game instructions.

```
print(“String to print”)

print(i)

print(backgroundsGroup.numChildren)

print(“The num of elements in the bg is “..backgroundsGroup.numChildren)
```

It in important to note that if you try to print an object (for instance, you save an image into a variable and print this variable), console will print the word “table” and a number. It doesn’t give you a lot of information, but it is still

helpful if you need to make sure that your object is not empty (is not nil).
Read more on print function here.

Also we need to talk about **comments** in Lua.

One-line comments in lua start with two dashes:

```
-- create a new Rectangle
```

To uncomment the line – remove the dashes

Multiline comments start with two dashes and two square brackets, and end with two square brackets:

```
--[[
create a new Rectangle
then add it to the touchGroup
]]
```

To uncomment multilines – remove dashes and brackets at the beginning of the comment, and brackets at the end of the comment.

Part 1. Project Architecture

This is the most important part of any development. It doesn't matter if you are developing something solo, or work in a big team. The clearer your project architecture is in your mind - the easier it is to develop.

Look at your main.lua. You'll see lots of comments explaining what should happen in a function, what is kept in a variable and so on. All of this information comes from project architecture. Because if you don't know what to do - you simply can not do it.

Let's start with the screen player sees when the game starts.

Screen1:

Player sees

A background image that covers all the screen.

Play button in the middle of the screen.

A name of a developer under the play button, also in the middle of the screen.

Player can push the button area. That will load the game scene shown on screen2

Screen2:

Before new screen loads

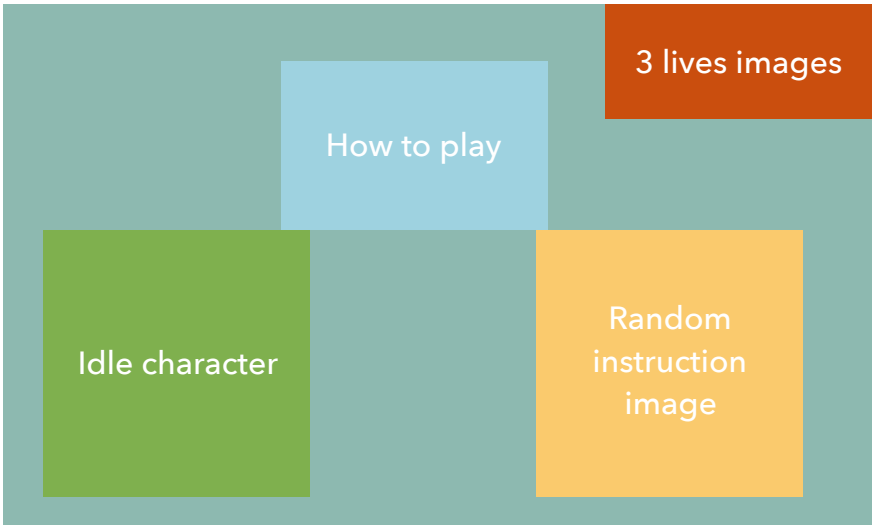
Background image is no longer visible.

Play button is no longer visible

Name of a developer is no longer visible

Area to push is no longer active.

Game clears stats and loads all the assets needed to display game stages



Player sees

A random background image

A character image in idle state

An informational image that shows how to play the game. It disappears after 1 second

Max number of lives (the same image repeated several times next to each other)

Random instruction for a swipe (1. direction is saved for comparison, instruction is counted)

Random soundtrack starts playing

From now until the end of the game the screen area captures player's swipes. When this top object captured a swipe, game state changes to Screen2_a

Screen2_a:

Before the screen changes

Player's gesture analysed (App1).

Result is calculated (App2)

Display changes according to result

App1

Check the direction of the swipe. Consider imprecise gestures.

Save the gesture direction and use it to calculate the result

App2

Check if it was the last instruction

if it wasn't, check if the gesture result is the same as one saved from instruction

the gesture is the correct one

display relevant character image, display new random instruction

the gesture is wrong - check if it is the last life

if it wasn't the last life - display character in the "wrong" position, remove one life, hide old lives images and redraw lives based on the current number of lives, display a random instruction

else - display character in the "fall" position, display informational image "fail", stop listening for swipes, hide lives images, stop audio. Display restart button.

if it was the last instruction

Hide lives, reset current lives and number of shown instructions, stop audio, stop listening for swipes.

Check if the gesture result is the same as one saved from instruction

if the result is right - display informational image "amazing", display our character in the relevant position.

else check if it was the last life

it wasn't the last life - display character in the "wrong"position, display informational image "decent"

it was the last life - display character in a "fall" position, display informational image "fail"

with a short delay display restart button in the middle of the screen, functionality of the button is described in App3.

App3

Restart button is an image displayed in the middle of the screen. It is loaded along with all assets, but made visible only when game ends.

When the button is displayed, all informational images are hidden, the button starts listening for the player’s touch.

When player touches the button, game reloads.

All assets were loaded at the beginning, so we don’t reload them again.

We remove the reload button and stop listening for player's touches on it.

Reset the stats.

Display a random background

Hide all character images and show only Idle one

Redraw lives with maximum number

Display a random instruction

Start playing a random musical composition

Start listening to the player’s swipes again

This isn't the most complete project architecture plan, but it accounts for the most important game actions, asset loading and game reactions to player-generated events. Now lets write some code!

Part 1. Corona specific concepts

To create our game we'll use several Corona and Lua specific things. Here's the list.

- Variables
- Tables
- Images
- Groups
- Texts
- Rectangles
- Events
- Listeners
- Transitions
- Timing Functions

During this workshop we'll look at each one of them separately. For now check out the top part of your main.lua file.

You'll see a number of variables declared outside of any function. We consider them **module variables** because they are available to any function in your mail.lua file

To keep our game progress, to get access to our images at any point in time, we need to keep them somewhere. For that we use local variables that are visible to the whole file and any function can access them at any time. You can read more on variables scope here.

All those variables are on top of your main.lua file now. Let's look through them and learn some new concepts.

Some of these variables are just declared (they have a name, but don't have a value), for example

```
local touchRect
local sound
```

Some are initialised, which means they a value assigned to them, for example

```
local INSTRUCTIONNUM = 15
local backgroundsGroup = display.newGroup()
local catImages  = {}
```

Here we have several different objects. **INSTRUCTIONNUM** is a number, **backgroundsGroup** is a group, **catImages** is a table.

If you have any programming experience, you may notice that Lua doesn't need to know a variable type when you declare it. It means that when you have declared a variable playButton that is visible for all functions, in any of those functions you can put an Image, Text, Integer or Audio Stream into it.

Bun then you initialise a variable (put a value into it) and if it is not a simple Integer, Float (decimal numbers) or String (a word), you'll have to use a constructor to "construct" a more complex object like Image or Text or Rectangle.

Different kinds of objects need different constructors. The group constructor is the simplest:

```
display.newGroup()
```

We'll look at constructors with parameters later.

Sidenote: Groups are very important in Corona. Here we are using only a small part of their functionality. Please, read more on groups here, when you have time.

Part 1. First attempts in graphics

Despite our game starting with main menu, we'll start with the more interesting part - our play scene.

We start with a function setPlayScene() that sets all of our assets on the screen. If you search through main.lua file you'll see a function setPlayScene() twice. Once where it is defined (where it has "function" word in front of it), and once at the end of the file. The second one is the call to this function that happens every time your game loads.

```
setPlayScene()
```

```
function setPlayScene()
      -- Reset stats
      setGameStats()

      -- load a background
      loadSceneBackground()

      -- load all the character poses into catImages
      loadAllCatPoses()
      hideAllCatPoses()

      -- display the idle character lose
      displayCat("idle")

      -- load all instruction images into instrImages
      loadAllInstructions()
      hideAllInstructions()

      -- display our first random instruction, save it into
                                          currentInstruction
      displayRandInstruction()

      -- load music by saving the value returned by the
      loadMusic() into sound variable and start playing music
      --[[
      sound = loadMusic()
      audio.play(sound)
      ]]

      -- create a swipe listener for player's   touches
      createSwipeListener()
end
```

Now let's look through **setPlayScene()** function.

It is a very long function that sets our scene. It calls many smaller functions to prepare the stage to play the game.

The first function is **setGameStats()**. This function is already in your main.lua file and the only thing it does is it resets the **instructionCount** to **0** to make sure we start our instructions from the beginning. Note it is not local, because we declared this variable on top of the package.

```
function setGameStats()
	instructionCount = 0;
end
```

Now lets work on graphics. Find the function **loadSceneBackground()**

```
function loadSceneBackground()
	-- Load background image
	-- Position it
	-- Resize it
end
```

Notice that every function in Lua has the word function in front of its name end the word "end" at the end. That allows Corona to know that that's where the function ends.

Let's write this function.

loadSceneBackground()

Images are created in Corona using this constructor:

```
display.newImage(group, path)
```

If you remember - we have a set of **group** created at the beginning of the file, and they are available to all functions.

The **path** is simply the location of the image in your project.

To create a background we need to store new image in a local variable. We can make it local to the function (invisible to all other functions), because we always can access it through the display group (our groups are available to all functions)

Read more on Image construction here.

So, create a local variable **background1** and initialise it using display.newImage with the group backgroundsGroup and image path "img/background1.png"

```
function loadSceneBackground()
	-- Load background image
	local background1 = display.newImage(backgroundsGroup,
"img/background1.png")

	-- Position it
	-- Resize it
end
```

When you save your file main.lua now, Corona Simulator will reload on its own and you'll see your background… in a very strange place.

Corona has a concept of anchors. They help you to align the image. Anchors can be accessed through object properties - **objectName.anchorX** and **objectName.anchorY**. You can read more about anchors here.

By default they are set to 0.5, which is the centre of the image. But when you load image into Corona, it aligns its centre with the top left corner of the screen.

And you see what you saw just now - a quarter of the image. To fix it you can either give the image coordinates to position itself on the screen - we'll do that later. Or to change it's anchors to the top left corner.

```
function loadSceneBackground()
	-- Load background image
	local background1 = display.newImage(backgroundsGroup,
"img/background1.png")

	-- Position it
	background1.anchorX = 0
	background1.anchorY = 0
	-- Resize it
end
```

Save the file and you'll see you image in the middle.

But depending on your device the image might not cover all the screen. We need to resize our background to cover all the screen. We can do that by setting its width and height to the width and height of the display that are available through **display.contentWidth** and **display.contentHeight** that we'll take from display object. You can read more on display sizes here.

```
function loadSceneBackground()
	-- Load background image
	local background1 = display.newImage(backgroundsGroup,
"img/background1.png")

	-- Position it
	background1.anchorX = 0
	background1.anchorY = 0
	-- Resize it
	background1.width = display.contentWidth
	background1.height = display.contentHeight
end
```

Important to note that we'll avoid "magic numbers" and try to use information about display width and display height as much as possible. Not just because it is a good practice, but also because this way your game will look pretty on any device.

Now you should see your background in the correct position.

loadAllCatPoses()

This function loads all the images with character poses.

```
function loadAllCatPoses()
      -- load each one character image separately and put it
      into catImages table with a relevant key
      -- Iterate through catImages table and position and
      resize each image

end
```

You already know how to create local variables and store images in them. So you'll need to create 8 variables (for each character image in the folder) and using a constructor display.newImage(group, path), load images into them.

For our character we'll use display group actorsGroup. It is also visible to every function. The names of each file you can see in your project folder. The example:

```
local catWin = display.newImage(actorsGroup, "img/
character/win.png")
```

That deals with images creation, but what about "putting them into a table" and why do we need a separate table for them, despite having a display group?

The simple answer is - we need a separate table for character images because we'll need to access them by their key. But let's first talk about tables a bit.

In Corona - everything is a table. If you don't believe me, print your background1 and you'll see.

But **tables** in Corona might be **indexed** (then they work like arrays) and each element in the table will be available through its index.

Here's the example.

tableFoodForToday = {"apple", "bacon", "pie"}

tableFoodForToday[1] will contain "apple", tableFoodForToday[2] will contain "bacon" and tableFoodForToday[3] will contain "pie"

Important! Table indices in Corona start with 1!

Or **tables** might contain **key-value pairs** (like maps or dictionaries in other languages):

local tableFoodForToday = {

breakfast = "apple",

lunch = "bacon",

dinner = "pie"

}

Then tableFoodForToday["breakfast"] will give you "apple" and so on.

Here we need a **key-value** table, because later we need to show an image where our character is "idle" or "fall".

To put an object into a key-value table we use the following syntax:

```
tableName[“key”] = objectToPutIn
```

Based on this example we can use the word in a file name as a key. For example:

```
local catWin = display.newImage(actorsGroup, "img/character/
win.png")

catImages["win"] = catWin
```

You can read more on Corona Tables here.

Do that for all character images, you can even copy the declaration, simply changing the variable name, file name, key and variable name again.

You should get something like this:

```
-- load each one character image separately and put it into
catImages table with a relevant key

      local catWin = display.newImage(actorsGroup, "img/
                        character/win.png")

      catImages["win"] = catWin


      local catIdle = display.newImage(actorsGroup, "img/
                        character/idle.png")

      catImages["idle"] = catIdle


      local catWrong = display.newImage(actorsGroup, "img/
                        character/wrong.png")

      catImages["wrong"] = catWrong


      local catFall = display.newImage(actorsGroup, "img/
                        character/fall.png")

      catImages["fall"] = catFall


      local catUp = display.newImage(actorsGroup, "img/
                        character/up.png")

      catImages["up"] = catUp


      local catRight = display.newImage(actorsGroup, "img/
                        character/right.png")

      catImages["right"] = catRight


      local catLeft = display.newImage(actorsGroup, "img/
                        character/left.png")

      catImages["left"] = catLeft


      local catDown = display.newImage(actorsGroup, "img/
                        character/down.png")

      catImages["down"] = catDown
```

Now we need to position and resize each element. We don't want to position them all separately, because they should end up in the same spot on the screen. So we use a loop to iterate through all character images and position them using 3 lines of code instead of 24 lines.

To loop (iterate) through the table with key-value pairs we use the following syntax.

```
for k in pairs(tableName) do
                -- do something with tableName[k]
end
```

k will give you each key in the group one after another, tablename with this key will give you the value. For example, if the first key is "win" – then catImages[k] is charWin object, because we just put it in under a key "win" into catImages. On the next iteration, k will be "idle" and the object - charIdle, and so on until the end of the table.

Inside the loop we need to position each of our images on the screen and resize it.

To position our image we'll use parameters x and y, and use display.contentWidth and display.contentHeight as a reference. For example - catImages[k].x we'll set to display.contentWidth/4, and it will align image centre with 1/4 of the display. catImages[k].y will be display.contentHeight/1.5 that will position the image in the lower part of the screen.

To resize the image we'll use the function **object:scale(1,1)**. The numbers here in the brackets are the percentage, use 1.0 for 100%, 2.0 for 200%, or 0.5 for 50%. You can read more on scale function here.

Did you notice that all the parameters of our object we used earlier were separated by dot, but **:scale** and some other functions are separated by colon? There's an explanation for that, and to put it simply - functions that you access with colon receive not only their parameters (1 and 1 in this example), but also the object itself as the first argument. Here's a very helpful article about it.

Now try to iterate through **catImages** and set all the values we discussed for each object in **catImages**. I recommend to scale them to 0.8 and 0.8 as well, but it's up to you.

```
for k in pairs(catImages) do
   catImages[k].x = display.contentWidth/4
   catImages[k].y = display.contentHeight/1.5
   catImages[k]:scale(0.8, 0.8)
end
```

hideAllCatPoses()

The next function call in setPlayScene() is to hideAllCatPoses()

```
function hideAllCatPoses()
      -- Iterate through catImages table and hide each one
end
```

You already know how to iterate through catImages. Now do it again, but instead of positioning and resizing each element, set its parameter **objectName.isVisible** to false. That will hide the object.

```
function hideAllCatPoses()

      for k in pairs(catImages) do

            catImages[k].isVisible = false

      end

end
```

Save your file again and you won't see any character images anymore.

displayCat(pose)

Here is something new. Before we only created functions without any parameters. This one is the first one that takes a parameter. If you look at the call to this function in **setPlayScene()**, you'll see that it looks like **displayCat("idle")**.

This happens because the function **displayCat(pose)** can be called with any character pose. “idle”, “fail”, “up” - at different points in our game we’ll need to display different character images.

We know that this function will be called with one of the keys from **catImages**. All we need to do is to find the image by its key and make it visible again.

```
function displayCat(pose)
      -- Make the relevant character pose visible using
      "pose" parameter as a key
      catImages[pose].isVisible = true
end
```

It is a potentially erroneous situation because if a more complex game you might end up calling the function with keys that do not exist. The good practice is to check if we have this key in our table, but our game is quite simple so we won’t bother with that.

That is all we need to do with our character.

Let’s move on to our instructions!

loadAllInstructions() and hideAllInstructions()

Find both of those empty functions. Do the remind you of anything? They are pretty much the same as instructions for loading and hiding character images. The difference will be in the file names, the name of the display group (it will be **instructionsGroup**) and the name of the table to put them in (**instrImages**).

Try to write them on your own and then compare with the following. A little reminder:

```
local imageName = display.newImage(group, address)

tableName[“key”] = objectToPutIn

object.isVisible = false

for k in pairs(tableName) do
       -- do something with tableName[k]
end
```

That's what you should get at the end:

```
function loadAllInstructions()
	local instUp = display.newImage(instructionsGroup,
				"img/icons/upSw.png")
	instrImages["up"] = instUp

	local instDown = display.newImage(instructionsGroup,
				"img/icons/downSw.png")
	instrImages["down"] = instDown

	local instLeft = display.newImage(instructionsGroup,
				"img/icons/leftSw.png")
	instrImages["left"] = instLeft

	local instRight = display.newImage(instructionsGroup,
				"img/icons/rightSw.png")
	instrImages["right"] = instRight

	for k in pairs(instrImages) do
		instrImages[k].x = display.contentWidth/1.3
		instrImages[k].y = display.contentHeight/2
		instrImages[k]:scale(0.8, 0.8)
	end
end
```

```
function hideAllInstructions()

	-- Iterate through instrImages table and hide each one

	for k in pairs(instrImages) do

		instrImages[k].isVisible = false

	end

end
```

displayRandInstruction()

The next function is displayRandInstruction(). Find it in the file. As you can see from the comments, we'll need to use a number of local variables. One to keep the names of the instructions, another one to store a random number. One more to keep the randomly chosen instruction name.

The first new local variable we'll need is **local i**. We'll save a random number into it. To get a random number, we'll use a function **math.random(min,max).** It takes 2 parameters - start and end value. For example math.random(1,4) will give you any number between 1 and 4 inclusive. We have only 4 directions in our game, so 1..4 is enough.

```
local i = math.random( 1, 4 )
```

Read more on math.random here.

In this function you'll see a table instructionKeys with the directions. All the directions will repeat our keys for the **instrImages** table. You can look them up in your code for that, or just believe me when I say the keys of instructions are "up", "down", "right" and "left". Uncomment this local variable.

```
local instructionKeys = {"up", "down", "right", "left"}
```

Here we work with an indexed table, because you don't see any keys in instructionKeys table, only values. But all those values can be accessed through the indices. First element has index 1, second - index 2, third - 3 and fourth - 4. Again, a reminder - indices in Lua start with 1.

We already have a random number from 1 to 4 stored in variable **i**. So if we create another local variable with a name **key,** and put one of the elements of instructionKeys table there, we won't know which of the words end up there. All we know that it will be one of those 4.

```
local key = instructionKeys[i]
```

Now we can use this key to get a random instruction from our **instrImages** and make it visible

```
instrImages[key].isVisible = true
```

We also need to do some housekeeping. Keep the key to compare it to the user's gesture later and count this shown instruction. Both will go into the package level variables that we have on top of the file - currentInstruction and instructionCount

```
function displayRandInstruction()
      local instructionKeys = {"up", "down", "right", "left"}
      local i = math.random( 1, 4 )
      local key = instructionKeys[i]

      instrImages[key].isVisible = true
      currentInstruction = key
      instructionCount = instructionCount+1
end
```

Part 1. Events and listeners

Start with createSwipeListener() function. Find it in your code. Then you can either uncomment its body to analyse it, or remove the inner part of the function and try to right it on your own. Look at the comments to see what we need to do.

```
function createSwipeListener()
      -- create a new Rectangle in the middle of the screen
      with the size of the screen, save it into touchRect
      -- add it to touchGroup display group
      -- position it in the middle
      -- change it's alpha to 0 to make it transparent
      -- set the parameter isHitTestable to true to make sure
      it still receives hits wile transparent
      -- add a "touch" event listener with a name
      playerSwipeEvent
end
```

There are several new concepts in this function. We'll start with a new type of object.

The constructor for the rectangle object:

```
display.newRect( group, x, y, width, height )
```

You can read more on rectangle object here. Our rectangle should be a module variable because we'll need access to it. We already have variable touchRect , we just need to put a new Rectangle into it. Group will be a touchGroup. X and Y we'll set to 0, because we already know how to position elements with anchors, and width and height should be the display size because this object should cover all the screen. Try to write it on your own, then compare your result with the function at the end of this block.

The next step is to position it in the middle with anchors. We already learned how to do it.

To access object's alpha use **objectName.alpha** and then set **objectName.isHitTestable** to true. The last part is where magic happens!

To detect the event, to know that the user had touched the screen, you need one of the objects on the screen to "Listen" for this touch. This object is called EventListener.

You can read more on Events and Listeners here.

To add an event listener to the object, use **objectName:addEventListener(type, eventFunctionName)** function that takes type of the event and name of the event function.

Let's talk more about events. Corona has lots of events – for example, if two objects collide in corona - that's an event. User actions in Corona are events. But we only need to listen for a "touch" event - that's when player touched the screen. The name of the event function will be **playerSwipeEvent**, and we'll write it right after this function. The event generated when user "touched" the screen will be given to **playerSwipeEvent** function as a parameter, and will contain a lot of information about the touch. We'll use some of it.

Check your result:

```
function createSwipeListener()
	touchRect = display.newRect(touchGroup, 0, 0,
		display.contentWidth, display.contentHeight )
	touchRect.anchorX = 0
	touchRect.anchorY = 0
	touchRect.alpha = 0
	touchRect.isHitTestable = true
	touchRect:addEventListener("touch", playerSwipeEvent)

end
```

Now we are going to work on playerSwipeEvent function.

playerSwipeEvent(event)

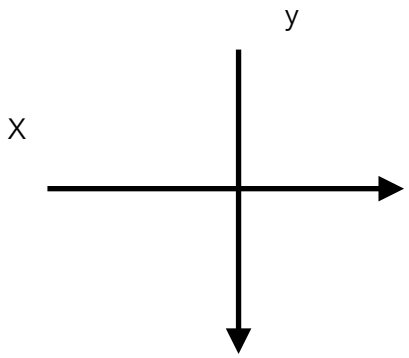
The function that processes the event receives it as a parameter. That's why the functions that deal with events always have the word (event) in brackets.

Every event in Corona has a phase. You can read more on "touch" event phases here. For our purposes the only phase we need is phase "ended". To check the event phase we'll use parameter event.phase:

```
	if event.phase == "ended" then

		-- gesture analysis
	end
```

We are going to analyse the player's gesture in this function, and as you can see from the comments - it's a bit complicated. Also a reminder - all the comments come from project architecture. If you look at the beginning of this text, you'll find an explanation of gesture analysis, but let's look at it again. Imagine device screen. Here are its axis:



X increases when user moves their finger to the right. Y increases when user moves their finger down.

To determine in what direction player's finger was moved, we can use coordinates of the beginning of the touch event and the end of the touch event.

- X coordinate of the beginning: event.xStart
- X coordinate of the event end: event.x
- Y coordinate of the beginning: event.yStart
- Y coordinate of the event end: event.Y

First, after checking the event phase, create a **local result** That's a temporary variable we'll use to keep the result of our comparisons In each of our 4 cases (x increased, x decreased, y increased, y decreased) we'll need to set result to the relevant string.

We can also print result to the console to make sure it works.

First, let's check whether x changed (was the gesture horizontal?)

```
if event.xStart-event.x>0 or event.x-event.xStart>0 then
	print(“Horizontal”)
else
	print(“Vertical”)
end
```

Then we’ll have an inner condition checking if x increased or decreased. In the first case, player’s finger moved to the left, so we need to save the string “left” into result variable.

```
	if event.xStart>= event.x then
		result = “left”
	else
		result = “right”
	end
```

That’s it for this inner part. Return to the first if.

```
if event.xStart-event.x>0 or event.x-event.xStart>0 then
	print(“Horizontal”)
	if event.xStart>= event.x then
		result = “left”
	else
		result = “right”
	end
else
	print(“Vertical”)
end
```

In else clause we'll have all the cases when x didn't changed (more on that later), and here we also need an inner if to check whether Y increased or decreased. If Y decreased - we are dealing with the swipe up, otherwise the swipe was down.

Also we need to print the result to the condole and get the result variable to the calculateResult function at the end.

This is how your function should look at the end:

```
function playerSwipeEvent(event)
  if event.phase == "ended" then
      local result
      if event.xStart-event.x or event.x-event.xStart then
            print("Horizontal")
            if event.xStart>= event.x then
                  result = "left"
            else
                  result = "right"
            end
      else
            print("Vertical")
            if event.yStart>= event.y then
                  result = "up"
            else
                  result = "down"
            end
      end
      print(result)
      calculateResult(result)
   end
end
```

Test it. And don't freak out when it works strange!

If you look at prints, you'll see that now all of our swipes are horizontal. That's because we process horizontal gestures first. So every gesture that wasn't a perfect vertical movement now is considered to be a horizontal one. The easiest way to fix it? Consider a swipe to be horizontal only if the difference between event.xStart and event.x is over 100 px, instead of 0px.

That's not a perfect fix, but good enough for now. Test your game again and look at prints.

```
if event.xStart-event.x>100 or event.x-event.xStart>100 then
```

Audio

Uncomment the call to loadMusic function and audio.play(sound), but as a courtesy either put your headphones on, or turn off the sound. You can enjoy testing the game music later.

For that go to setPlayScene() and uncomment lines:

```
        --[[
        sound = loadMusic()
        audio.play(sound)
        ]]
```

You can look at the explanation for this function in part 2, but this should make your music play

To start audio you'll need to use function **audio.play(sound)** that receives a sound variable. To stop it you need to use function **audio.stop()**

calculateResult(result)

Another function with a parameter. And from the looks of it it's quite long, but if you read comments you'll see we have all of those functions already written, including music. So we just need to call them! Use audio.stop() to stop the music where needed.

```
function calculateResult(result)
	hideAllInstructions()
	hideAllCatPoses()
	if instructionCount<INSTRUCTIONNUM then
		if result==currentInstruction then
			displayCat(currentInstruction)
			displayRandInstruction()
		else
			displayCat("fall")
			audio.stop()
		end
	else
		touchRect:removeEventListener("touch", playerSwipeEvent)
		audio.stop()
		if result==currentInstruction then
			displayCat(currentInstruction)
		else
			displayCat("fall")
		end
	end
end
```

Follow the comments and add calls to the relevant functions, then check with this file.

And that's it. Now your little game should work. And if you want to develop it further - follow up with Part 2. There we'll create main menu, give player some feedback about his or hers actions, randomise backgrounds and will create a restart function for our game.

Part 2

First we need to stop our game from loading level 1 from the beginning. To do that find a line **setPlayScene()** at the end of your **main.lua** file and remove it, or comment it out. If your save your game, after that it will load with an empty black screen.

For the second part of the game we'll need some additional module variables. We'll need an additional group to put our main menu background into. And another group for our HUD elements (lives and text images). Add them to the list of your display groups, but think for a bit about the order. The main menu background will be the first thing that appears in the game, so **mainBackgroundGroup** should be declared first. **hudGroup** should follow **instructionsGroup**.

```
-- Display groups for organizing the level.
local mainBackgroundGroup = display.newGroup()
local backgroundsGroup = display.newGroup()
local actorsGroup = display.newGroup()
local instructionsGroup = display.newGroup()
local hudGroup = display.newGroup()
local touchGroup = display.newGroup()
```

We will also need variables for the buttons.

```
-- Variables that keep the game control
local playButton
local restartButton
local authorName
local touchRect
local sound
```

For lives and text images we'll need additional tables. Declare two tables **lifeImages** and **textImages** in the same space where we declared our tables for instruction images and cat images.

```
-- Tables to keep sets of images
local catImages  = {}
local instrImages  = {}
local lifeImages  = {}
local textImages = {}
```

Additional constants will be **ANIMTIME** (number of milliseconds for animations - 1000) and **LIVESNUM** (maximum number of lives a user can have - 3). Those constants should be initialised and set to values that we won't be changing.

```
-- Static variables that won't be changed by game
local INSTRUCTIONNUM = 15
local LIVESNUM = 3
local ANIMTIME = 1000
```

We will also need a variable to keep the number of lives user currently has. We can initially set it to the same number as **LIVESNUM**.

```
-- Variables to keep game stats
local instructionCount = 0
local livesCurrent = 3
local currentInstruction = 'idle'
```

Now when we have our variables, let's start working on the menu.

We need to write functions to load our main menu. We'll need to create Menu Background, Menu play button and Menu Author text. We already declared variables for this images and this text, now we'll initialise them in our functions.

Write 3 empty functions and call them at the end of your file instead of setPlayScene()

```
setMenuBackground()

setMenuPlayButton()

setMenuAuthorText()

— — setPlayScene()
```

setMenuBackground() and setMenuPlayButton()

```
function setMenuBackground()

      -- create a local  mainbackground  variable and put a
      new image into it with a group mainBackgroundGroup

      -- set its anchors to 0

      -- resize it to the display hight and width

end
```

You already know how to do it all, because it generally repeats the creation of background for the game.

```
function setMenuPlayButton()

      -- create a new image for the hudGroup and save it into
      buttonBack variable

      -- put it in the middle of the screen

      -- add an event listener playTouchEvent with type
      "touch" to it

end
```

If you have difficulty aligning your button with the middle of the screen, just remember that you need to align its x and y to the centre. And the centre equals to the half of display width, and the half of display height.

Also, if you add a touch event using playTouchEvent listener and save your game - it will reload with an error. Because there's no such function, we haven't written it yet. To fix it simply write en empty event function **playTouchEvent** (remember it will get an event as a parameter). We'll fill it out later.

setMenuAuthorText()

```
function setMenuAuthorText()
      -- create a local variable that contains a set of
      options for the text, the parent group is hudGroup
      -- experiment with x and y to get the right position
      -- create a new Text object with those options and save
      it into authorName
      -- change text colour using
      objectName:setFillColour(0,0,0)
end
```

Here we are going to do something new. We are creating a text object. The constructor for the text object is:

```
display.newText( options )
```

To create a text you'll need a table of options. You can read more on the parameters for options here, we'll use only some of them.

For our text we need a parent group, the text itself, x and y coordinates, width, the font, size and alignment.

```
local options =
      {   parent = hudGroup,
          text = "Developed by – Your Name–”,
          x = ?
          y = ?
          width = 800,
          font = native.systemFont,
          fontSize = 40,
          align = "center"
      }
```

Lets talk about alignment of this text. It's X coordinate should be the center of the display (**display.contentWidth/2**). But if we also align in to the center of the display by Y (**display.contentHeight/2**), it will cover the button playButton we've created earlier. There's an easy fix for that. Our **playButton** is the package level variable, and we made sure that we created **authorName** text only after we created the **playButton**. So we can use the width of our **playButton** to position our text right under it.

```
local options =
      {
      …
      x =  display.contentWidth/2,
      y = display.contentHeight/2+playButton.width/2,
       …
      }
```

After we created our local set of options, we are ready to initialise **authorName** with a Text constructor. **:setFillColor(1,1,1)** uses the RGB values from 0 to 1, you can experiment with different colours here, starting from 1, 1, 1. I set mine to 0.6,0.9,0.9

Here's the example of the code you should get to create your main menu. Don't forget to write an empty event function, your code won't work without it.

```
-------------------------------------------------------
-- Set the main menu.
-------------------------------------------------------
function setMenuBackground()
      local mainbackground = display.newImage(mainBackgroundGroup,
      "img/openBackground.jpg")
      mainbackground.anchorX = 0
      mainbackground.anchorY = 0
      mainbackground.width = display.contentWidth
      mainbackground.height = display.contentHeight
end



function setMenuPlayButton()
      playButton = display.newImage(hudGroup, "img/icons/
      playButton.png")
      playButton.x = display.contentWidth/2
      playButton.y = display.contentHeight/2
      playButton:addEventListener( "touch", playTouchEvent )
end

function playTouchEvent(event)


end
```

```
function setMenuAuthorText()
      local options =
            {
                parent = hudGroup,
                text = "Developed by Inga Pflaumer",
                x = display.contentWidth/2,
                y = display.contentHeight/2+playButton.width/2,
                width = 800,
                font = native.systemFont,
                fontSize = 40,
                align = "center"
            }
      authorName = display.newText(options)
      authorName:setFillColor(0.6,0.9,0.9)
end
```

The last thing we need to do to for our menu to start working is to fill out the **playTouchEvent(event)** function. Here's what you need to do in it:

```
function playTouchEvent(event)
      -- Check the event phase
            -- make the main background invisible, getting access to
            it trough the display group (it is the only member)
            -- make the playButton invisible
            -- make the authorName invisible
            -- remove playTouchEvent from playButton
            -- call function setPlayScene()
end
```

If you remember, in our **setMenuBackground()** we created a local variable named **mainbackground**. Because it it local we are unable to get access to it and mare it invisible through **mainbackground.isVisible = false**.

You can fix it by declaring **mainbackground** variable as a package level variable (to do that you need to move it up ti the place we declared other package level variables like display groups or tables. But you can also use some Corona magic. At the beginning of the workshop we learned that groups are very helpful. And this is an example of that. You know that our **mainbackground** is in the group mainBackgroundGroup and that it is the only object in this group. So you can access it any time through the index 1.

So your **mainbackground.isVisible = false** will transform into:

```
mainBackgroundGroup[1].isVisible = false
```

All other actions in **playTouchEvent(event)** are pretty straightforward so write it on your own and then compare to this result:

```
function playTouchEvent(event)
      if event.phase=="ended" then
            mainBackgroundGroup[1].isVisible = false
            playButton.isVisible = false
            authorName.isVisible = false
            playButton:removeEventListener( "touch",
            playTouchEvent)
            setPlayScene()
      end
end
```

Try it - your main menu should load and when you push the button - your game should start.

Now let's make it more interesting using a random background (we have 5 of them, so we can randomly show one when game starts).

Working with backgrounds

Before we start randomising our background, let's comment out the call to

loadSceneBackground() inside our **setPlayScene()** function. And add calls to 3 new functions:

loadBackgrounds(), **hideBackgrounds()** and **displayRandomBackground()**

As you can guess, they do pretty much the same thing as earlier load, hide and display cat functions, or our load, hide and display instructions functions.

loadBackgrounds() should do the following:

```
function loadBackgrounds()
      -- create 5 local variables of image type to keep 5
      background images, add them into display group
      backgroundsGroup
      -- iterate through the backgroundsGroup setting the
      anchors to 0 and width and hight to display width and
      height
end
```

Before we iterated through tables with key-values pairs, and here we are going to iterate through a display group. That can be done in a simple for loop, starting with i=1 and iterating to the number of children in the display group. This parameter can be accessed as:

```
backgroundsGroup.numChildren
```

So, here's what your for loop should look like:

```
      for i=1, backgroundsGroup.numChildren do
            -- do something with backgroundsGroup[i]
      end
```

hideBackgrounds() should again iterate through **backgroundsGroup** and hide all elements.

Now let's look at **displayRandomBackground()** function.

We need to create a local variable i and put a random value from 1 to 5 into it and display a background with that index. Try to do it on your own and then compare with the code below:

```
function  loadBackgrounds()
	local background1 = display.newImage(backgroundsGroup, "img/background1.png")
	local background2 = display.newImage(backgroundsGroup, "img/background2.png")
	local background3 = display.newImage(backgroundsGroup, "img/background3.png")
	local background4 = display.newImage(backgroundsGroup, "img/background4.png")
	local background5 = display.newImage(backgroundsGroup, "img/background5.png")
	for i=1, backgroundsGroup.numChildren do
		backgroundsGroup[i].anchorX = 0
		backgroundsGroup[i].anchorY = 0
		backgroundsGroup[i].width = display.contentWidth
		backgroundsGroup[i].height = display.contentHeight
	end
end


function hideBackgrounds()
	for i=1, backgroundsGroup.numChildren do
		backgroundsGroup[i].isVisible = false
	end
end


function displayRandomBackground()
	local i = math.random(1, 5)
	backgroundsGroup[i].isVisible = true
end
```

And that is it - now each time you reload your game you'll get a random background.

Lives - checks and displays

Again we'll need to load Lives, hide Lives and display Lives in relation to the current number of lives.

Create 3 empty functions **loadLives()**, **hideLives()** and **displayLives()**. Insert calls to them into your **setPlayScene()** function. Let's look at what we need to do in our **loadLives()** function.

loadLives()

```
function loadLives()
	-- create a local variable and put an image of the heart into it with a group hudGroup
	-- insert this image into lifeImages using the function table.insert function
	-- repeat this for all 3 lives images
	-- position each image separately, their Y is the same live1.height and their x is based on live1.width
end
```

The main difference between loadLives() and other loading functions we already have is the positioning. All the cat images are positioned in the same spot. But for lives we need 3 images side by side. Also we won't always have 3 images, we might have 2 images if player made a mistake, 1 image if player made 2 mistakes and no images at all if player made 3 errors and plays the last life.

Because of that we need to code positioning of the life on the screen and in the table correctly, so we never get into a situation when player has 2 lives of 3 but we see the first heart and the last heart with a hole between them.

```
table.insert(tableName, index, objectName)
```

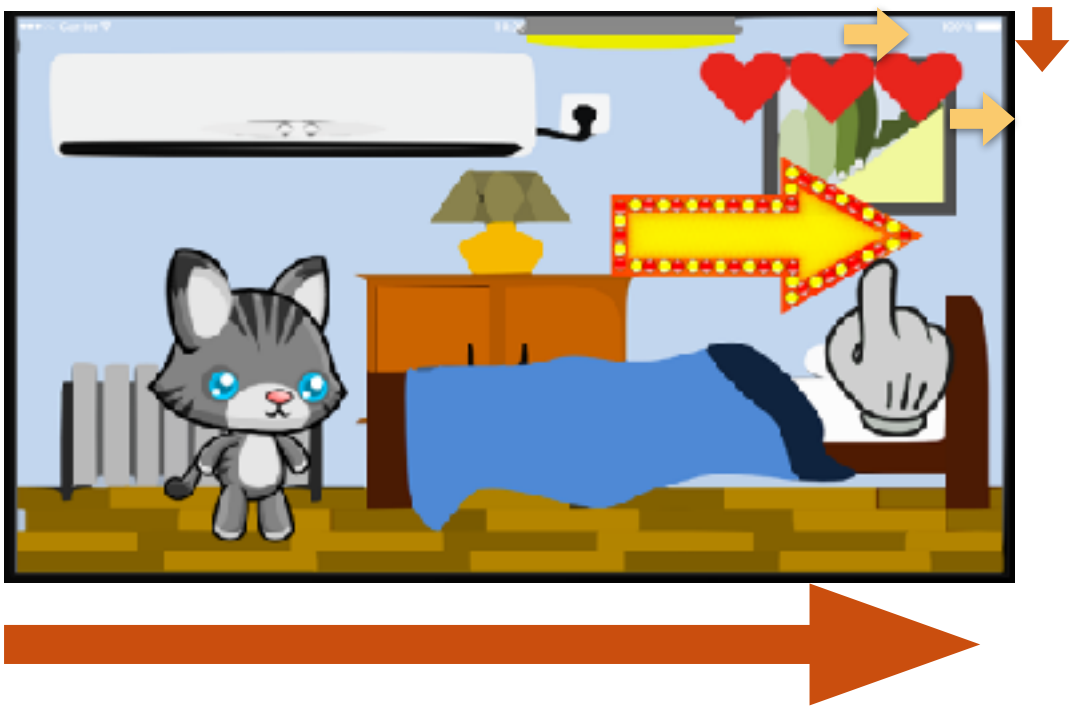
To do that we'll use an indexed table and objects are added to indexed tables in a slightly different way.

So, you need to create a local variable and save a heart image into it, adding it into **hudGroup,** and then insert the same object into **lifeImages** table. Don't forget that indexes start from 1, not 0!

Repeat this step 3 times using the same image address (because all hearts look the same), and then position each one separately. How do you find a position for each element?

Let's think geometrically. We need to put the first one of the hearts at the right edge of the screen. To do that instantly we can use display.contentWidth for X (moves the image to the right) and 0 for Y. But that looks a bit ugly, so let's move it away a bit from top and right edges. We don't want to hard code indent in pixels, but we can use one of the object parameters, for example - the width of the image itself.

Set **live1.x** to display.contentWidth and subtract live1.width from it to move the image away from the screen side. Because the image object is aligned by it'c centre, you'll see that the first heart is moved away from the screen side to the half of life image width. Then instead of setting **live1.y** to 0, set it to the **live1.height** and that will also move the image the half of its width down.



That's coordinates for the first image. But what about the second and the third one?

Again think geometry. We know images are aligned by their centre and they have the same size. So to position the second heart in the same space you need to subtract image width from the **display.contentWidth** and then to move it further to the left, you'll need to subtract another **live1.width** - this will move the second heart to the side of the first heart.

For the heart number 3 you'll need 3 subtractions.

So, that's what you should get:

```
function loadLives()
	local live1 = display.newImage(hudGroup, "img/icons/live.png")
	table.insert(lifeImages, 1, live1)
	local live2 = display.newImage(hudGroup, "img/icons/live.png")
	table.insert(lifeImages, 2, live2)
	local live3 = display.newImage(hudGroup, "img/icons/live.png")
	table.insert(lifeImages, 3, live3)

	live1.x = display.contentWidth - live1.width
	live1.y = live1.height
	live2.x = display.contentWidth - live1.width*2
	live2.y = live1.height
	live3.x = display.contentWidth - live1.width*3
	live3.y = live1.height
end
```

hideLives() and displayLives()

Here we need to iterate through **lifeImages** table and hide all images. You'll need to iterate from 1 to LIVESNUM because that's the maximum number of lives we can have.

In **displayLives()** you'll need to iterate again, but this time up to **livesCurrent**, because we might have 2, 1, or even 0 lives (in the last case

there won't be any hearts on the screen). Don't forget to set the visibility of the image in **lifeImages[i]** to true.

That's how these 2 functions should look like:

```
function hideLives()
	for i=1, LIVESNUM do
		lifeImages[i].isVisible = false
	end
end
function displayLives()
	for i=1, livesCurrent do
		lifeImages[i].isVisible = true
	end
end
```

If you try to play your game right now, you won't see the changes in lives, because at this point our game is played only until the first error. To fix that we need to rewrite **calculateResult()** function, but before that let's add informational images so our player knows how to play a game, and we would learn some interesting things about transitions.

Informational images

Lets look at informational images in our icons folder. We have 4 informational images. 3 of those assess the player's actions. For this prototype we'll only show them at the end of the level (when player has completed the last gesture, or lost all lives). We need to show "Fail!" image if there're no lives left, "Decent!" image if player ended the level with the wrong gesture, but still has lives left, and "Awesome!" image if the last gesture was the final correct one. You can later play with conditions and show images for every gesture, or based on lives number or anything else.

Again, we'll need 3 functions: **loadAllInfoImages()**, **hideAllInfoImages()** and **displayInfoImage(key)**.

loadAllInfoImages()

```
function loadAllInfoImages()
	-- Load new images with text information right into textImages array with individual keys
	-- Iterate through key-value pairs and put each image in the middle of the screen by X and in the first quarter of the screen by Y
end
```

First, create an empty function with this name and call it inside **setPlayScene()** function. Now let's write this function and try a different approach. You know that we need to load all the images into **textImages** table, and because we need to show one specific image at a time - it should be a key-value pairs table.

Earlier we did the following sequence: Load images into variables -> Save them into table under a specific key.

This approach, though it is visual, is also not optimal. When dealing with a table we can actually skip the first step. So instead of creating a local variable, we can save the image into a table cell right away.

The syntax is:

```
tableName["key"] = display.newImage(group, location)
```

So the first step in **loadAllInfoImages()** will be to load all images into the **textImages** table under a descriptive key, adding the images to **hudGroup** and moving them to the correct location.

Then you'll need to iterate through textImages (do you remember how to iterate through key-value table?) and position each element. Try positioning it in the middle of the screen on X axis and 0.25 of the screen on Y axis.

hideAllInfoImages() and displayInfoImage(key)

```
function hideAllInfoImages()
	-- Iterate through textImages array using key-value
	parts and
	-- Make each element invisible my making its alpha 0
end

function displayInfoImage(key)
	-- Take the image with this key from textImages. And
	set its alpha to 1.
	-- Then using Corona function transition.to(object,
	{parameter, time}) change its alpha back to 0 in
	ANIMTIME miliseconds
end
```

First one iterates through **textImages** and hides all elements. But instead of simply making every element invisible, setting its **isVisible** to false, try instead making in transparent by setting its alpha to 0. We did it earlier for the rectangle that captures player's gestures, so you know how to do it.

Don't forget to call **hideAllInfoImages()** right after you load all images in **setPlayScene()**.

Second one gets a key of the image and displays this image, setting its alpha to 1. But there's a catch. We don't really need those informational images on the screen all the time. So what should we do to hide them over time? That's where transitions come in.

Transitions are wildly used in Corona in animations and interactions. You can read more on transitions here.

The syntax for transitions:

```
transition.to( object, {parameter, time})
```

The target will be our **textImages[key]**, but what about parameters? There's a number of parameters that can be changed in transition, but for

our purposes we only need two - parameter to change and the time to change it.

```
transition.to(textImages[key], {alpha=0, time=ANIMTIME})
```

So in the first line we will set **textImages[key].alpha** to 1 and then slowly fade it away for ANIMTIME miliseconds (animation time is one of our constants, we declared it on top of our file and it is visible to all functions).

Now add a call to **displayInfoImage(key)** in **setPlayScene()** function and **key** will be the key that you gave to the informational image that explains how to play.

So, overall that's where you should be with these functions:

```
function loadAllInfoImages()
	textImages["awesome"] = display.newImage(hudGroup, "img/
				icons/awesome.png")
	textImages["decent"] = display.newImage(hudGroup, "img/icons/
				decent.png")
	textImages["fail"] =  display.newImage(hudGroup, "img/icons/
				fail.png")
	textImages["info"] =  display.newImage(hudGroup, "img/icons/
				swipe.png")

	for k in pairs(textImages) do
		textImages[k].x = display.contentWidth/2
		textImages[k].y = display.contentHeight/4
	end
end
```

```
function hideAllInfoImages()
	for k in pairs(textImages) do
		textImages[k].alpha = 0
	end
end


function displayInfoImage(key)
	textImages[key].alpha = 1
	transition.to(textImages[key], {alpha=0, time=ANIMTIME})
end
```

Now, to make sure everything is in order, check your setPlayScene() function. It should look something like this:

```
function setPlayScene()
	-- Reset stats
	setGameStats()
	--loadSceneBackground()
	-- Part2: load all backgrounds
	loadBackgrounds()
	-- Part2: hide all backgrounds
	hideBackgrounds()
	-- Part2: display a random one
	displayRandomBackground()
	-- Part1: work with a Cat
	loadAllCatPoses()
	hideAllCatPoses()
	displayCat("idle")
	-- Part2: load lives images, display max LIVESNUM
	loadLives()
	hideLives()
	displayLives()
	sound = loadMusic()
	audio.play(sound)


(continues on the next page)
```

```
	-- Part1: load all instruction images into instrImages
	loadAllInstructions()
	hideAllInstructions()
	displayRandInstruction()
	-- Part1: create swipe listener
	createSwipeListener()
	-- Part2: load all info images into textImages, display and
hide instruction
	loadAllInfoImages()
	hideAllInfoImages()
	displayInfoImage(“info")
end
```

Audio

Before we get to the results, let's look at our audio function **loadMusic()** You know all the concepts used there. I need to note that what we are doing here is not the optimal way of working with sounds, but it works. With all that you know now, I am sure you will be able to write it on your own. The main thing is that our **loadMusic()** returns the loaded audio stream, so we can save it into a module variable **sound** in **SetPlayScene()** and then play it and stop it when we want to. You can read more on audio manipulations in Corona here. And here's the function:

```
function loadMusic()
	-- create a local variable i in the range from 1 to 3
	local i = math.random(1,3)
	-- create a local indexed table musicNames with strings that
	contain names of compositions in sound folder
	local musicNames = {"track1.wav","track2.wav","track3.wav"}
	-- create a local variable soundtrack, take one composition
	from the musicNames with index i and use it as a name of file
	in audio.loadSound function. Set {loops = -1,} when you
	loading your sound to loop it
	local soundtrack = audio.loadSound("sounds/"..musicNames[i],
		{loop=-1})
	-- return this soundtrack variable so you can use it anywhere
	later
	return  soundtrack
end
```

New calculateResult(result) function

Now we need to rewrite our **calculateResult(result)** function to take into account the lives that player has at any point. What and where should we add?

```
function calculateResult(result)
	hideAllInstructions()
	hideAllCharPoses()
	if instructionCount<INSTRUCTIONNUM then
	-- It wasn't the last instruction - game continues
		if result==currentInstruction then
		-- It was the right gesture so we don’t need to check lives
			displayCat(currentInstruction)
			displayRandInstruction()
		else
		-- Wrong gesture so we need to se if player can continue the game
			-- if it wasn't the last life (livesCurrent>0)
				-- Display character in the "wrong" position
				-- Remove one life, hide lives
				-- display lives again because their number changed
				-- Display another random instruction
			-- else if was the last life player had
				-- Display character in the "fall" pose
				-- Part2: Display info image "fail"
				-- Remove the listener from the touch rectangle
				-- hide lives images because we'll need to redraw
				-- stop audio
				-- Part2: display Restart button
			end
		end
	(continues on the next page)
```

```
else
-- It wasn the last instruction so game ends either way
              hideLives()
              touchRect:removeEventListener("touch", playerSwipeEvent)
              audio.stop()
              if result==currentInstruction then
              -- It was the right gesture so we don’t need to check lives
                     displayCat(currentInstruction)
                     -- Part2: Display info image "awesome"
              -- else (the result is not the same as currentInstruction)
              else
                     -- if it wasn't the last life (livesCurrent>0)
                            -- Display character in the "wrong" position
                            -- Part2: Display info image "decent"
                     -- else (it was the last life)
                     else
                            displayCat("fall")
                            -- Part2: Display info image "fail"
                     end
              end
              -- Part2: using Corona function timer.performWithDelay, call
              displayRestart() function with a delay for ANIMTIME miliseconds
       end
end
```

Try to fill out all the blanks on your own until you get to the last comment. Here you have another new concept (last concept for this workshop) - delayed functions. You can read more on delayed functions here.

```
timer.performWithDelay(delay, functionName)
```

Here we need a short delay for player to see the result of his game before trying again.

The syntax for delayed functions is as follows:

```
timer.performWithDelay(ANIMTIME, displayRestart)
```

We need to delay our restart for ANIMTIME time. Please note that performWithDelay here takes a function name without brackets.

displayRestart()

Before we can display our restart button, we need to write an additional function **createRestartButton()** so there is something to display. Create an empty function **createRestartButton()** and call it in your **setPlayScene()** to make sure that this button is created. To keep up with our structure, we actually need to create 2 functions - one to display and one to hide the restart button, but to save time we'll do it inside one function. Generally it is not a good idea to combine two functionalities inside one function, but we can get away with it here.

```
function createRestartButton()

	-- create a new Image (image is in your img files
	folder), save it into restartButton (we created it as a
	package value at the beginning of part 2). Put this
	image into hudGroup

	-- position it in the centre of the screen

	-- make it invisible

end
```

From here you can guess what **displayRestart()** should do, but again – let's make it more functional. We know that we need to display restart button on top of everything else, and that when player sees it they need to know that game ended, so Instructional images with gestures should be hidden. Also we need to add en event Listener to the button for actual restarting the game.

```
function displayRestart()

	-- hide all informational images

	-- make restartButton visible

	-- add a reloadGameEvent listener of "touch" type to the
restartButton

	restartButton:addEventListener("touch", reloadGameEvent)

end
```

Let's get to the last part of the game - restarting the game.

reloadGameEvent(event)

At this point you might be tempted to simply call **setPlayScene()** here. That would be a mistake, because our **setPlayScene()** actually sets all the objects that we need. So if we load them all again - our game will grow uncontrollably. It is always a good idea to separate asset loading from showing the assets (you can see it every time we write 3 functions - for loading, hiding and showing).

We know that all the objects we need are already on our scene. All we need to do - hide them, remove the event listener from the button we just pushed, and add a listener to the big rectangle that covers our screen and captures player's gestures (we removed it earlier to make sure we only capturing a touch on the restart button when the game ended).

```
function reloadGameEvent(event)

	-- check the event phase

		-- remove the reloadGameEvent from the restartButton

		-- make restartButton invisible again

		-- reset your game stats

		-- display a random scene background

		-- hide the character images

		-- display an Idle character

		-- Display lives

		-- Display a random instruction

		-- Clear the sound variable by setting it to null

		-- Load a new composition into loadMusic() and save it
		into sound variable

		-- play music

		-- Add a playerSwipeEvent with type "touch" to the
		touchRect

end
```

Almost there. Now we need to change the function that resets our stats. Why? Because now we have new stat value - **livesCurrent** and we need to set it back to LIVESNUM, as player starts with maximum number of lives. Add the line **livesCurrent = LIVESNUM** to your **setGameStats()** function, and now your game will be restarting with the right number of lives.

And that is it. Your first game is ready. To test in on an actual device look through an article on Provisioning and Building on iOS or Signing and Building – Android.

Thank you for participation! Any feedback will be highly appreciated.