School of Science

CSP2108 Introduction to Mobile Application Development

Workshop: Using files for Olympic Training

In this workshop you will complete a simple app for training on Olympic Games knowledge. Most of the code is provided, and shows how you can separate code into multiple source files, gives example code for creating scenes, playing audio, introducing time delays, and responding to user interactions (clicking on buttons).

Your part is to supply the code to read Olympic Games knowledge from a text file, so that the program can use it to devise randomized questions and answers.

Instructions

Download OlympicQuiz.zip from Blackboard and unzip it to a suitable location. This is the Corona project. Try it out. It works, but currently has a very small knowledge base, which is hard-coded. It looks like this:

We'll come to your part soon, but first, we will take a look at the existing code.

Code in multiple files: require()

Look in the project folder and you should find (amongst other things) main.lua, questionsScene.lua and questions.lua, as well as olympicData.txt and two audio files, buzzer.wav and chime.wav.

This app is a bit more complicated than those we've worked on so far, so the code has been split across the 3 .lua files. This helps to manage the complexity of the code and promotes code re-use: each file has responsibility for some coherent aspect of the program. In this case:

- main.lua – as usual this is the starting point for the app. It is quite short and looks after orchestrating all the other parts.
- questionsScene.lua – in this app, there is only one user interface "scene", in which a question is presented to the user (e.g. "Which was the Olympic city in 1920?") and a set of possible answers is displayed using 4 buttons (e.g. "Helsinki", "Los Angeles", "Sydney" and "Antwerp"). The user clicks on the button that they think has the right answer. The program tells the user if they are right or not, and then loads the next question. The logic for how this scene works is in this source file.
- questions.lua – this is a kind of "utility" module. The code in here knows nothing about apps or user interfaces. What it does is to create a "knowledge base" of Olympic knowledge, and provide a set of methods for accessing that knowledge. questionScene.lua uses these methods, without needing to know anything about how the knowledge is created and managed.

This is done in Lua using the require() function, which allows one chunk of Lua code to "load in" another chunk when needed.

Take a look at main.lua:

```
-----------------------------------------------------------------------------------------
--
-- main.lua
--
-- main for OlympicQuiz example
--
-----------------------------------------------------------------------------------------

-- even though we only have one scene, we will use the composer library

local composer = require( "composer" )

-- Code to initialize the app can go here
```

```
-- Now load the opening scene

-- Assumes that "questionScene.lua" exists and is configured as a Composer scene
composer.gotoScene( "questionScene" )
```

Here require() is being used to load in the *composer* library – one of Corona's libraries, whose job it is to manage "scenes". Scenes are used in Corona like different screens in a traditional GUI program. The *composer* library lets you separate your user interface into a number of scenes, which you can switch between as the user does different things with your app. We will learn more about this library in Week 8.

Some other Corona libraries are "built-in", so you don't have to use require() to load them in (but you can if you like).

Further down, the call to composer.gotoScene() is also going to use require() – behind the scenes - to load in questionScene.lua .

Notice that near the start of questionsScene.lua we also have the line:

```
local composer = require( "composer" )
```

because questionsScene.lua also users the composer library, so it looks like *composer* is going to be loaded twice, but require() is smart about how it works, and will only load once, no matter how many times it is called for the same code.

You will also find, a bit further down, this line:

```
local questions = require( "questions" )
```

This loads in the code from questions.lua. If you look at the code for questions.lua, you will see that it starts with:

```
local questions = {}
```

and ends with

```
return questions
```

The first line creates a table, the code in between creates local variables and functions, as well as table properties and methods, and the bottom line returns that table. When this code is loaded using require(), a reference to the table is returned and this is stored in the local variable *questions* in questionsScene.lua. The code in questionsScene.lua can then access the properties and methods in the table.

For example look at this function:

```
-- go on to next question
local function nextQuestion( event )
	questions:setNextQuestion()
	question.text = questions:getQuestionText()
	question:setFillColor(1, 1, 1)
	for i = 1,4 do
		answers[i]:setLabel(questions:getAnswer(i))
	end
end
```

This function is called when a new question is needed. You can see that it calls 3 methods on *questions*: setNextQuestion(), getQuestionText() and getAnswer().

The composer library

We have already mentioned the *composer* library for creating and maintaining "scene"s. This app has only one scene, the code for which is in questionsScene.lua. This code was created by starting with the example Scene Template from

https://docs.coronalabs.com/guide/system/composer/index.html .

We then added some code of our own for loading in audio files, creating text and buttons, and handling interactions with the user. Most of the work in building the user interface has been added to the method scene:create(). This method is called by the *composer* library the first time that the scene is needed, so it is a good place to put the user interface building code. The *composer* library handles all the business of moving user interface items onto the screen when a scene is loaded, and off again when another scene takes its place.

The other method we have added code to is scene:destroy(), which is called when the scene is no longer needed. In our app we use this to tidy up the resources we used for the audio.

User interaction with Widgets

This is the first app we have looked at which lets the user interact with it. We will learn about how this is done in next week's lecture, but here we can have a sneak preview. The code for this is in questionsScene.lua.

Take a look at makeAnswer :

```
-- utility to make a Widget for one of the possible answers
local function makeAnswer( ID, x, y )
```

```
	answer = widget.newButton(
			{
				label = questions:getAnswer(ID),
				onEvent = handleAnswer,
				emboss = false,
				-- Properties for a rounded rectangle button
				shape = "roundedRect",
				x = x,
				y = y,
				width = answerWidth,
				height = answerHeight,
				cornerRadius = 2,
				fillColor = answerFillColor,
				strokeColor = answerStrokeFillColor,
				strokeWidth = 4
			}
			)
	return answer
end
```

This function creates a Widget (button) that shows one of the possible question answers. You can specify the text (label), location (x,y) and size (width, height) and so on, similar to other DisplayObjects. The new part here is that you can specify a function to be called when the user does something with the button, such as clicking on it. This is called an *event handler* – in this example, its name is "handleAnswer". The code for handleAnswer() is just above in the code :

```
-- if the user has selected their answer, check it and give feedback, then go to the next question
local function handleAnswer( event )
	if questions:checkAnswer(event.target:getLabel()) then
		question.text = "Correct!"
		question:setFillColor(0, 1, 0)
		audio.play(soundTable["yesSound"])
	else
		question.text = "No! " .. questions:getCorrectAnswer()
		question:setFillColor(1, 0, 0)
		audio.play(soundTable["noSound"])
	end
	-- Wait a couple of seconds before the next question
	timer.performWithDelay(2000, nextQuestion)
end
```

When the user clicks on the button, this function is called, and an *event* object is passed as a parameter, providing information about what happened. In this case *event.target* will be the object that was clicked on, so we can use that to find out which answer the user clicked on, and see if they got the right answer.

The line:

```
timer.performWithDelay(2000, nextQuestion)
```

is another example of using an event handler. In this case, a 2 second (2000 ms) timer is set, and the function nextQuestion() is called when the timer goes off, giving the user enough time to see whether they got the right answer before the next question is loaded.

Notice that there is a problem here : the logic will be messed up if the user clicks again on one of the Widgets before the timer goes off. We won't try to fix that – just to keep the code simple for explanation purposes.

Now for your bit: reading questions from a data file

As it stands, the app works, but it only has 6 facts loaded into its knowledge base, and these are hard coded in questions.lua :

```
data[1] = { year = 1896, city = "Athens", country = "Greece" }
data[2] = { year = 1900, city = "Paris", country = "France" }
data[3] = { year = 1904, city = "St Louis", country = "USA" }
data[4] = { year = 1908, city = "London", country = "UK" }
data[5] = { year = 1912, city = "Stockholm", country = "Sweden" }
data[6] = { year = 1920, city = "Antwerp", country = "Netherlands" }
```

This is a bit of a clumsy way to get the knowledge in. Knowledge is really a kind of data, not program logic. It would be better to have the app read this data in from a file or a database. This will be your job today – replace the code above with a chunk of Lua code that reads the Olympic Games facts from a text file, and loads them into the *data* table.

To do this, you will have to write code to open a text file, read lines from the file, and process each line using Lua's string functions. You should find most of what you need to know in this week's lecture slides, or the Corona string manipulation guide at https://docs.coronalabs.com/guide/data/luaString/index.html or the Corona reading and writing files guide at https://docs.coronalabs.com/guide/data/readWriteFiles/index.html .

Comment out the above hard coded lines, and put in new code that does the following steps:

1. Find the path to "olympicData.txt". Hint: this file will be in the *system.ResourceDirectory* .
2. Open the file for reading (your code should handle any errors here).
3. If the file was successfully opened, then
    a. Read the file one line at a time (Hint: use a for loop)
    b. Extract the year, city and country data from each line (Hint: use *match()* from the string library, with a suitable string pattern)
    c. Add a new array element to *data*, with the year, city and country.
    d. Close the file

e. Don't forget to set the file variable to nil.

Hint for the pattern to use with match():

You will want a pattern that has 3 captures in it, so something that looks like

(xxx)yyy(zzz)www(vvv)

where you have to figure what to put for xxx, yyy, zzz, www and yyy.

xxx should match a string of digits (this will be the year)
yyy should match everything between the year and the start of the city, including the "," and any spaces
zzz should match the city, which can have letters and spaces
www should match everything between the city and the start of the country, including the "," and any spaces
vvv should match the country, which can have letters and spaces