Intro to Corona SDK

Haeyong Chung
Spring 2017

Slides contain examples from the official Corona docs as well as the textbook.

Corona SDK Online Reference

- Corona SDK has an excellent online documents/reference:

https://docs.coronalabs.com

Review: Real Quick Look at Lua

- Three Types of Variables
  - Global
  - Local
  - Tables
- String Concatenation
- Logical Operations

Review: Assign Values to Variables

```
foo, bar = "WOW", "YES“
print(foo) -- WOW
print(bar) -- YES
print(foo, bar) – WOW   YES
```

Review: Global

- Global
  - No need for declaration

```
myVariable = 10
print( myVariable ) -- prints the number 10
```

Review: Local

- Local
  - Local scope limited to a function or a block (similar to most common languages of today)
  - It needs to have a separate declarative assignment using
  - the keyword "**local**" on the first assignment
    - What is the result of print(x)?

```
x = 10 -- global 'x' variable
local i = 1
while i <= 10 do
    local x = i * 2 -- a local 'x' variable for the while block
    print( x ) -- 2, 4, 6, 8, 10 ... 20
    i = i + 1
end
print( x ) -- ???
print ( i ) -- ???
```

Scope of local i

Scope of local x

Review: Table

- Groups of variables uniquely accessed by an index. (e.g., Array)
- Index can be numbers and strings or any value pertaining to Lua.
- Some different ways to create tables.

Review: Create Table (Continued)

- Indexed with Numbers (like an usual array):

```
Simpsons = {"Homer", "Marge", "Bart", "Lisa",
"Maggie"}  --just sequential order
```

Or (equal to)

```
Simpsons = Simpsons = { [1] = "Homer", [2] = "Marge",
[3] = "Bart", [4] = "Lisa", [5] = "Maggie" }  --
assign specific number to each element.
```

But it is possible for us to assign specific number to each one using this approach:

```
Simpsons = Simpsons = { [1] = "Homer", [200] =
"Marge", [4] = "Bart", [40] = "Lisa", [5] = "Maggie"
}
```

Review: Create Table (Continued)

- Indexed with Strings or characters:

```
Simpsons = {a="Homer", b="Marge", c="Bart", d="Lisa", e="Maggie"}
or
Simpsons = {};
Simpsons['a'] = "Homer"
Simpsons['b'] = "Marge"
Simpsons['c'] = "Bart"
Simpsons['d'] = 'Lisa'
Simpsons['e'] - 'Marge'
or
Simpsons = { a="Homer")  -- should not EMPTY BRACES!!!
Simpsons.b = "Marge"
Simpsons.c = "Bart"
…
or
Simpsons = {)
INDEX = "a"
Simpsons[INDEX]="Homer"  -- this is the same as Simpsons['a'] = "Homer"
and Simpsons.a = "Homer"
…
```

Review: Access to Table Elements

- All properties can be accessed using the dot operator (x.y) or a string (x["y"]) to index into a table

- Genera Array Type (Numeric Index):

```
Simpsons = {"Homer", "Marge", "Bart", "Lisa",
"Maggie"}  --just sequential order
Print(Simpsons[3]) -- Bart
```

- String or Characters as Indexes:

```
Simpsons = {a="Homer", b="Marge", c="Bart",
d="Lisa", e="Maggie"}
print (Simpsons.c)  --?
print (Simpsons[c]) --?
```

Review: Access to Table Elements (Continued)

- One more case…

```
Simpsons = {}
INDEX = "c"
Simpsons[INDEX]="Bart"

-- print Bart
print(Simpsons.INDEX)  -- correct?
```

Review: And Others…

- String Concatenation

```
print('hello' ..  ' world) -- hello world
print('hello' ..  1) -- hello1
```

  - But…

```
Print ('100' + 1 )  -- 101
```

- Logical Operators
  - Everything except nil and false is true!!!
    - x and y
      - returns x (1st arg) if the value is false or nil
      - returns y (2nd arg) otherwise

```
print (false and 'hello') -- false
```

    - x or y
      - returns x (1st arg) if the value is true.
      - returns y (2nd arg) otherwise

```
print (false or 'hello') - false
```

Pop Quiz (extra credit; no lost point)

1. Create a table which include five string elements (“one”, “two”, “three”, “four”, and “five”) indexed by ‘a’, ‘b’, ‘c’, ‘d’, and ‘e’ respectively.
2.
```
--[[
      print(UAH)
   ]]
```
3.
```
---[[ -- result?
      print(UAH)
  --]] -- result?
```
4. `print ("1"..1)  -- ?`
5. `print ("1" + 1) -- ?`

```
foo, bar = "WOW", "YES“
```
6. `print(foo) -- ?`
7. `print(bar) -- ?`
8. `print(foo, bar) -- ?`
9. `print("This is ‘a’ string!") -- ?`

```
Name1, Phone = "John Doe", "123-456-7890"
Name2 = "John Doe"
```
10. `print(Name1, Phone) -- ?`
11. `print (Name1 and Phone)  -- ?`
12. `print (Name1 or false) -- ?`
13. `print(Name1 == Phone) -- true or false?`

Pop Quiz (extra credit; no lost point)

1. Create a table NUMBER which include five string elements (“one”, “two”, “three”, “four”, and “five”) indexed by ‘a’, ‘b’, ‘c’, ‘d’, and ‘e’ respectively.

```
NUMBER = {a=“one”, b=“two”, c=“three”, d=“four”, and e=“five”}
```

2.
```
--[[
        print(UAH)
  ]] – no action
```
2.
```
---[[
  print(UAH)
  --]] -- UAH
```
3.
```
print ("1"..1)  -- 11
```
4.
```
print ("1" + 1) -- 2
```

```
foo, bar = "WOW", "YES“
```
6.
```
print(foo); -- WOW
```
7.
```
print(bar); -- YES
```
8.
```
print(foo, bar); -- WOW     YES
```
9.
```
print("This is ‘a’ string!") -- This is ‘a’ string!
```

```
Name1, Phone = "John Doe", "123-456-7890"
Name2 = "John Doe"
```
10.
```
print(Name1, Phone) -- John Doe         123-456-7890
```
11.
```
print (Name1 and Phone)  -- 123-456-7890
```
12.
```
print (Name1 or false) -- John Doe
```
13.
```
print(Name1 == Phone) -- false
```

Creating Multidimensional Arrays

- First Way:

```
local x = {{1, 2, 3} , {4, 5, 6}};
print (x[1][1],x[1][2],x[1][3]);
print (x[2][1],x[2][2],x[2][3]);
```

- Second Way:

```
local y = {};
y[1] = {1, 2, 3};
y[2] = {4, 5, 6};
print (y[1][1],y[1][2],y[1][3]);
print (y[2][1],y[2][2],y[2][3]);
```

Creating Multidimensional Arrays (continued)

```
1. local z = {};
2. local cnt = 1;
3. for i=1,3 do
4.    for j=1,3 do
5.           z[i][j] = cnt; -- ?
6.           cnt = cnt + 1;
7.    end
8. end
```

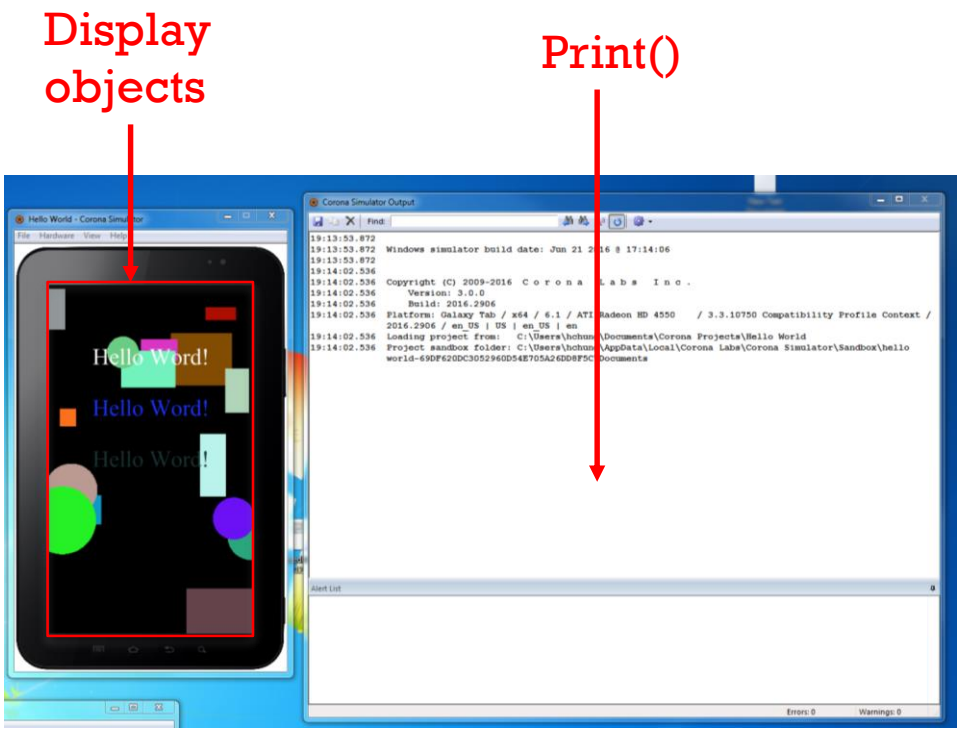
This doesn't work.

Creating Multidimensional Arrays (continued)

```
local z = {}; -- create first dimension
local cnt = 1;
for i=1,3 do
  z[i] = {}; -- create second dimension
  for j=1,3 do
    z[i][j] = cnt;
    cnt = cnt + 1;
  end
end
```

Corona: Display Objects

- Anything you put on the screen is done via display objects
  - Standard Image
  - Text
  - Animated Sprite
  - Rectangle
  - Circle
  - Rounded Rectangle
  - Line
  - Polygon
  - Display Group



- We can create and show the above by calling methods:
  - e.g., `display.newCircle()` will create a circle which is an instance of the display object.

Display Group

- Group objects are a special type of display object.
- In a display group, you can add/remove display objects as children of the group.
- In this case, the display group becomes a ***parent*** of those display objects.

```
local group = display.newGroup()
local rect = display.newRect( 100, 100, 50, 50 )
rect:setFillColor( 0.7 )
group:insert( rect ) --add rect to group
rect.parent:remove( rect ) --removes rect from group
```

Display Properties

- Instances of ***display*** objects behave in a manner similar to Lua tables.

- You can change the following properties of display objects:
  - **object.alpha:** This is the object's opacity. A value of 0 is transparent and 1.0 is opaque. The default value is 1.0.
  - **object.height** and **object.width**: decide width and height of the screen.
  - **object.isVisible**: Controls whether the object is visible on the screen (true or false).
  - **object.isHitTestable**: Allows an object to continue to receive hit events even if it is not visible. If true, objects will receive hit events regardless of visibility; if false, events are only sent to visible objects. It defaults to false.
    - true: even if the object is not visible, it will receive hit events (touching w/ finger)
    - false by default
  - **object.rotation**: the current rotation angle (in degrees).

Display Properties (Continued)

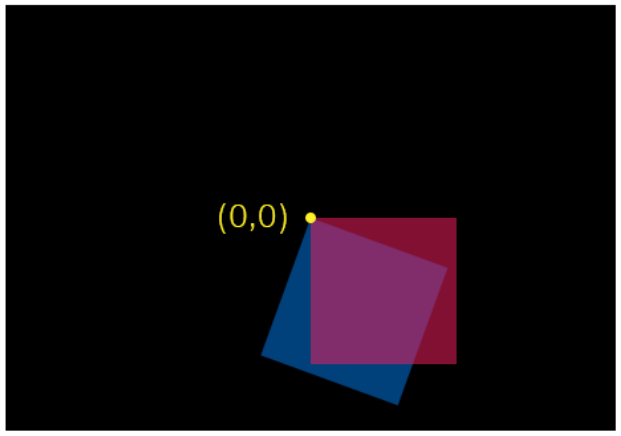
- **object.parent***:returns the parent object
- **object.contentBounds***: Table containing the boundaries: xMin, yMin, xMax, yMax
- **object.contentHeight*** and **object.contentWidth***: : This is the height and width in screen coordinates. This is similar to object.height or object.width except that its value is affected by y scaling and rotation.
- **object.x** and **object.y**: This specifies the x and y position (in local coordinates) of the object relative to the parent -- the parent's origin to be precise.
- **object.xScale** and **object yScale**: Change scale of the object in either direction (While scaling the visual size, it will NOT change how the physics engine perceives the object)
- **object.contentCenterX** and **object.contentCenterY**: the conter of content area along x and y.

*: read only

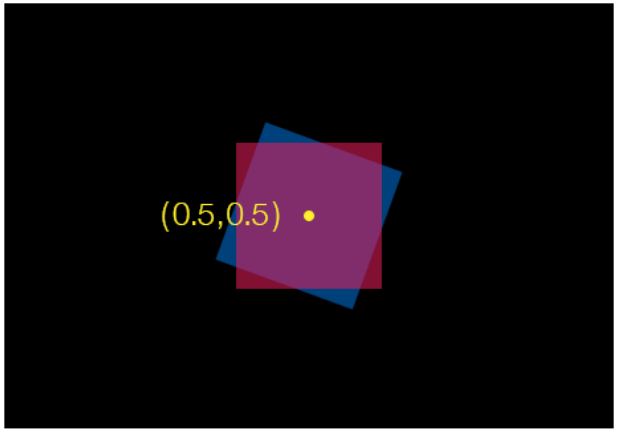
Display Properties (Continued)

- **object.anchorX** and **object.anchorY:** This specifies the x and y positions of the object's alignment to the parent's origin.
  - Alignment of object relative to the (x,y) position;Values : ***0=left , 0.5=middle, 1=right***

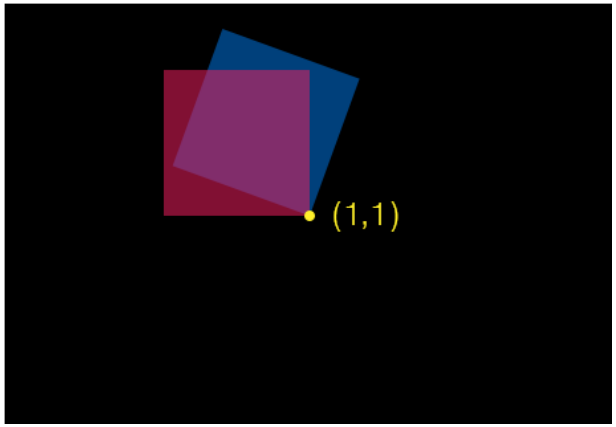
0=left



0.5=middle



1=right



Display Methods

- Several methods can be called by display objects
- There are two ways this can be done: using the dot operator (“.”) or using the colon operator (“:”).
  - Examples of the dot operator
    - object = display.newRect(110, 100, 50, 50)
    - object.setFillColor(1.0, 1.0, 1.0)
    - object.translate(object, 10, 10)
  - Examples of the colon operator
    - object = display.newRect(110, 100, 50, 50)
    - object:setFillColor(1.0, 1.0, 1.0)
    - object:translate(10, 10)

Syntactic Sugar: Dot vs. Colon

- Basically dot (.) and colon (:) operators are the same.
- However, the colon operator is just a syntactic sugar.
  - function object:method(arg1, arg2) is same as function object.method(self, arg1, arg2).
  - i.e., the colon is for implementing methods that pass self as the first parameter. So x:bar(3,4)should be the same as x.bar(x,3,4).

Display Methods (Continued)

- Display objects share the following methods:
  - **object:rotate(deltaAngle)** or **object.rotate(object, DeltaAngle)**:adds deltaAngle (in degrees) to the current rotation property.
  - **object:scale(sx, sy)** or **object.scale(object, sx, sy):** multiplies the xScale and yScale properties using sx and sy, respectively.
  - **object:translate(deltaX, deltaY)** or object.translate(object, deltaX, deltaY): This effectively adds deltaX and deltaY to the x and y properties respectively. This will move the object from its current position.
  - **object:removeSelf()** or **object.removeSelf(object)**: This removes the display object and frees its memory, assuming that there are no other references to it.

Image

- Use `display.newImage( [parentGroup,] filename [, baseDirectory] [, x, y][,isFullResolution] )`
- and return an image object (something like `imageObject = display.newImage('charger.jpg', 100, 100)`.
- Display objects support Autoscaling (by default) but you can manually turn on /off authoscaling by using isFullResolution parameter (false: autoscaling, true: no autoscaling)
- If you reload the same image multiple times, the subsequent calls to display. newImage ignore the isFullResolution parameter and take on the value passed the first time.

Place Images on Mobile Screen

- Let's create a code which show a background image and three other display objects of the same image.

1. Run the Simulator and Create a new project
2. Prepare two different image files (one foreground and one background)
3. Create a new main.lua

```
local sky=display.newImage("bkg_clouds.png")
local ground= display.newImage( "ground.png", 160, 445
)
local crate1= display.newImage( "crate.png" )
crate1.x = 180;
crate1.y = 80
local crate2 = display.newImage( "crate.png" )
crate2.x = 180;
crate2.y = 160
crate2.rotation = 10
```

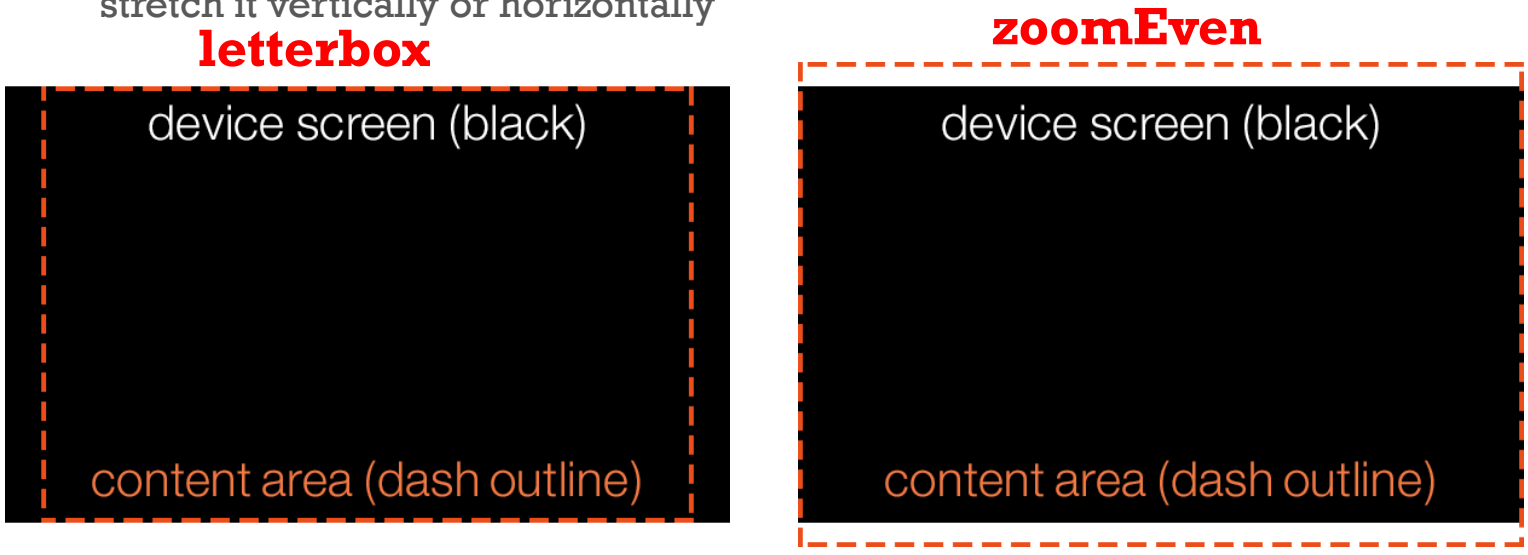
Runtime Configuration

- **Config.lua** allows you to Dynamic content scaling, dynamic content alignment, dynamic image resolution, frame rate control, and antialiasing

- Dynamic Content Scaling
  - width (number): This is the screen resolution width of the original target device (in portrait orientation)
  - height (number): This is the screen resolution height of the original target device (in portrait orientation).
  - scale (string): This is a type of autoscaling from the following values:
    - letterbox: This scales up content uniformly as much as possible
    - zoomEven: This scales up content to uniformly to fill the screen, while keeping the aspect ratio
    - zoomStretch: This scales up content nonuniformly to fill the screen and will stretch it vertically or horizontally

Runtime Configuration (continued)

- Dynamic Content Scaling
  - width (number): This is the screen resolution width of the original target device
  - height (number): This is the screen resolution height of the original target device
  - scale (string): This is a type of autoscaling from the following values:
    - letterbox: This scales up content uniformly as much as possible
    - zoomEven: This scales up content to uniformly to fill the screen, while keeping the aspect ratio
    - zoomStretch: This scales up content nonuniformly to fill the screen and will stretch it vertically or horizontally



Scale

Original Image

Letterbox

"letterbox"

ZoomEven

"zoomEven"

ZoomStretch

"zoomStretch"

Runtime Configuration (Continued)

- Dynamic Content Alignment: Content is dynamical aligned based on different devices.
  - xAlign: specifies the alignment in the x direction. The following values can be used (e.g., `xAlign = "left"`):
    - `left`
    - `center` (**default**)
    - `right`
  - yAlign: specifies the alignment in the y direction. The following values can be used:
    - `top`
    - `center` (**default**)
    - `bottom`

Runtime Configuration (Continued)

- Dynamic Image Selection : Automatically swap in higher resolution versions of your images to higher resolution devices

Runtime Configuration (Continued)

- Dynamic Content Alignment: Automatically swap in higher resolution versions of your images to higher resolution devices
  - Add @2x suffix to the end of filename (e.g., myImage@2x.png)
  - In your project config.lua file, a table named imageSuffix needs to be added for the image naming convention and image resolutions to take effect

```
application =
{
        content =
        {
                width = 320,
                height = 480,
                scale = "letterbox",
                imageSuffix = {
                        ["@2x"] = 2,
                },
        },
}
```

  - In our main.lua, you have call your display objects by using display.newImageRect, instead of displaynewImage().

Runtime Configuration (Continued)

- Another Config.lua Example:

```
application =
{
    content =
    {
      width = 320,
      height = 480,
      scale = "letterbox",
      xAlign = "left",
      yAlign = "top"
    },
}
```

Runtime Configuration (Continued)

- Frame Rate Control: you can decide a target frame for different devices (30 fps by default but you can set it to 60fps)

Shapes

- Create shapes (vector objects) such as rectangles, circles, and rounded rectangles using these methods:
  - **display.newRect([parentGroup,] x, y, width, height):** This creates a rectangle using width by height.
    display.newRoundedRect([parentGroup,] x, y, width, height, cornerRadius): This creates a rounded rectangle using width by height.
  - **display.newCircle([parentGroup,] xCenter, yCenter, radius)**: This creates a circle using the radius centered at xCenter, yCenter.

Shapes

- Style methods
  - **object.strokeWidth:** This creates the stroke width in pixels
  - **object:setFillColor(red, green, blue, alpha):** We can use the RGB codes between 0 and 1. The alpha parameter, which is optional, defaults to 1.0
  - **object:setStrokeColor(red, green, blue, alpha):** We can use the RGB codes between 0 and 255. The alpha parameter, which is optional, defaults to 1.0

Text

- display.newText([parentGroup,] text, x, y, font, fontSize)
  - Fonts:
    - Font
      - Enter font name, 'New Times Roman'
      - native.systemFont
      - native.systemFontBold
    - Color and String Value
      - Size: object.size
      - Color: object:setFillColor(red,gree,blue,alpha)
      - Text: object.text – this allow you to update a sting value

Change the Appearance of the Status Bar

- change the appearance of your status bar using the **display.setStatusBar(mode)** method

In-Class Exercise

- display.contentWidth and display.contentHeight gives you how big the screen is.
- divide the screen up into a 3x3 grid of your choice and automatically generate rectangles of different colors.
  - math.random() -- [0,1] real
  - math.random(m) -- [1,m] integer
  - math.random(m, n) -- [m, n] integer
- (This is not a graded homework, but similar to what needs to be done for HW1 later)