Learning Lua & Loving It !

Marc Garneau • Surrey, BC, Canada • T³ National Instructor
piman@telus.net • *http://web.me.com/piman2/PimanNspire/Blog/Blog.html*
T³ International Conference • Chicago, IL • March 3, 2012

Note: The documents in this handout can be found at: http://piman.ca/Lua/chicago.zip

Over the past few years I have been very active in developing TNS documents that engage students in meaningful mathematical inquiries. Having often pushed the TI-Nspire built-in functionality and programming to its limits (if not beyond), I was very excited to hear of the integration of Lua scripting into the TI-Nspire. What could I do with Lua that I couldn't do before? That is a question I'm still uncovering answers for, and that will continue. I feel very much still at an early phase in my own Lua understanding, but I am loving the process!

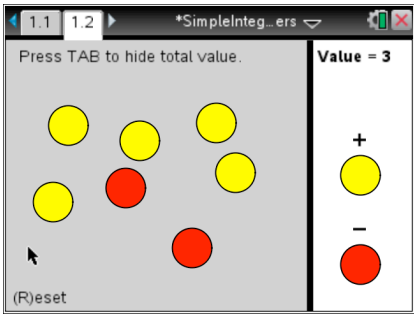
I have no formal programming background, just what I've picked up myself over the years, mostly learning from what others have done. Lua was brand new to me. I highly recommend beginning with Steve Arnold's tutorials. These have given me a strong foundation that I can now build from and learn some more on my own, as well as make sense of the wonderful scripts that have already been written!

http://compasstech.com.au/TNS_Authoring/Scripting/index.html

The focus of this session will be on using **classes** and **tables** to do something cool - something I couldn't do on the TI-Nspire without Lua:

As a context for introducing classes, I created a Lua script for modeling integers using counters. I've done this type of thing in a TNS before, but to do it, I had to create each and every circle! And for each circle, there were variables and programming elements, eg: for counting its value, resetting its position, colour, etc.

With this Lua version, each circle (counter) gets created when it's needed! The number of circles is theoretically limitless! Resetting to the beginning doesn't move circles back, it simply vanishes them from existence!
Below I provide the script, along with comments.



Note: *As I'm a newbie, there may be better ways to do these things, or better ways to explain what I've done.*

```
Color = {
        red = {0xFF, 0x00, 0x00},
        yellow = {0xFF, 0xFF, 0x00},
        }
```

This looks like a list of lists. It's actually what in Lua is called a **table**. The "red" and "yellow" are index names that can be used to refer to their definitions. Look for Color[...] references later.

```
function reset()
```

A Lua script can have many functions defined within it.
This *reset* function sets things back to *initial* settings.

```
    Objects = {
        Circle(xstart, ystartred, W/20, "red",-1),
        Circle(xstart, ystartyellow, W/20, "yellow",1),
    }
```

Objects is a table of instances of the class Circle. Its parameters will make more sense once we look at the Circle class later. The table begins with two circles constructed, which are the red and yellow circles that will appear on the right of the screen. New circles are constructed by adding more instances of the Circle class to this table. When this reset function is called, the Objects table is returned to its initial state of having just two elements.

```
    z = 0
```

z will be used to toggle whether the total value shows or not. Starting with 0 means the value won't be showing at the beginning.

```
end
```

end closes off the function definition.

```
function on.resize()
    W = platform.window:width()
    H = platform.window:height()

    xstart = 7*W/8
    xmat = 3*W/4
    ystartyellow = H/2
    ystartred = 5*H/6

    TrackedObject = nil

    reset()
end
```

on.resize(width, height) is one of the *event handlers* that are customized for the TI-Nspire integration of Lua. Whenever the application window changes size (eg: computer view vs. handheld view, etc.), this function will be called.

W and *H* are global variables for the particular window's width and height.

These define (x, y) positions of interest on the screen. The 'start' variables define starting points for the counters. The *xmat* variable defines where the mat ends.

Each time we grab a counter, it will become our tracked object. Initially, none of the counters are selected.

When the screen gets re-sized, also run the reset program to set things back to their initial values.

```
Circle = class()
```

A **class** is kind of a "super function" which can create instances of *class objects* with a set of defined self-associated attributes/behaviours, etc. That's probably a bad definition, but it's what I have in mind.
Here we define a class called *Circle* which will define and create the instances of circles (counters).

```
function Circle:init(x, y, radius,color,value)
    self.x = x
    self.y = y
    self.radius = radius
    self.color = Color[color]
    self.selected = false
    self.value = value
end
```

When each instance of *Circle* is initialized, the circle object (counter) has the following attributes:
- (x, y) coordinates
- radius (will be the same for all in this activity)
- color (selected from the Color table above)
- initially is not a selected object
- value (+1 for positive counter; -1 for negative counter)

```
function Circle:contains(x, y)
    local r = self.radius
    local d = math.sqrt((self.x - x)^2 + (self.y - y)^2)
    return d <= r
end
```

If our pointer (x, y) is inside the circle, then this function returns *true*, a fact which we'll make use of later.
Note that math.sqrt means $\sqrt{\ }$. This is from a Lua standard library of functions.

```
function Circle:paint(gc)
    local cx = self.x - self.radius
    local cy = self.y - self.radius
    local diameter = 2*self.radius
    gc:setColorRGB(unpack(self.color))
    gc:fillArc(cx, cy, diameter, diameter, 0, 360)

    gc:setPen("thin","smooth")
    gc:setColorRGB(0, 0, 0)
    gc:drawArc(cx, cy, diameter, diameter, 0, 360)

    if self.selected then
        gc:setPen("medium","smooth")
        gc:setColorRGB(0, 0, 0)
        gc:drawArc(cx, cy, diameter, diameter, 0, 360)
    end
end
```

As we'll see later, *paint* literally means to paint the screen (objects, text, etc.). *setColorRGB* sets the colour as specified for the particular circle.
gc:fillArc(x, y, width, height, startAngle, endAngle) is what we use to draw a filled circle. Note from its syntax that we could fill a partial arc, and that arc may be elliptical in dimension. You'll see a lot of *gc*. It's a dummy variable but well chosen to stand for *graphics context*.

gc:drawArc(x, y, width, height, startAngle, endAngle) is similar to *fillArc*, except it will just outline the circle. setPen specifies the style & thickness. The colour is set to black.

If the circle is selected (we'll see what triggers that later; see the mouseDown function), then use a thicker line to outline the circle.
if self.selected then is an interesting syntax. It means that if *self.selected = true* then do what follows.
Note that unlike TI-Nspire programming, *if* ends with *end* instead of *endif.*

```
function count()
    value = 0
    for i = 1, #Objects do
        local obj = Objects[i]
        if obj.x < xmat then
            value = value + obj.value
        else
            obj.x = xstart
            if obj.value == 1 then
                obj.y = ystartyellow
            else
                obj.y = ystartred
            end
        end
        if obj.x < 0 then
            obj.x = obj.radius
        end
        if obj.y > H then
            obj.y = H-obj.radius
        end
        if obj.y < 0 then
            obj.y = obj.radius
        end
    end
end
```

I initially set up the *count* function to figure out the total value of the integer counters, but because I was looping through the Objects table, it allowed me to do a bit more too.

Start with a value of 0.

If we were programming in TI-Nspire, this *for* loop would begin with *For i, 1, dim(Objects).* It means we'll be cycling through each element of the *Objects* table, and *obj* represents the individual element (which in this case is each instance of the Circle class object, ie. each circle/counter).

If the *x*-coordinate of the circle is to the left of *xmat*, ie. on the mat, then add its counter value (+1 for yellow, -1 for red).

If the *x*-coordinate of the circle is not to the left of *xmat*, then send it to its initial position on the right of the screen.
Note that we need to use a double-equal sign. A single-equal sign is used to define variables only.

If the circle gets dragged off the mat, then send it back onto the mat.

```
function on.mouseDown(x,y)
    for i = 1, #Objects do
        local obj = Objects[i]
        if obj:contains(x, y) then
            TrackedObject = obj
            obj.selected = true
            if obj.x == xstart then
                if obj.y == ystartred then
table.insert(Objects, Circle(xstart, ystartred, W/20, "red",-1))
                else
table.insert(Objects, Circle(xstart, ystartyellow, W/20, "yellow",1))
                end
            end
            platform.window:invalidate()
```

We now get to our *mouse event handlers*. This function gets triggered upon clicking the mouse down, or by clicking and holding for a second on the handheld.

Which object gets selected and tracked (acted on) depends on which object *contains* the cursor when the click down happens.

This is the coolest part of the whole script! If I select a circle (ie. grab a counter) from the 'stack' on the right, it will create a new one to replace itself! The *table.insert(table name, value)* command is used to add a new value to the end of the table/list.

The *platform.window:invalidate()* command gets used often. We use it usually whenever we need to refresh the screen. What I find the best way to see what difference this command makes is to comment it out, and then notice the difference, if any. There are situations where it's needed on the handheld even if it isn't needed when using the software.

```
            break
        end
    end
end
```

The *break* command is here because once we have identified the selected circle/counter, there's no need to keep looping through to test the other circles.

```
function on.mouseUp(x,y)
    if TrackedObject ~= nil then
        TrackedObject.selected = false
    end
    TrackedObject = nil
    platform.window:invalidate()
end
```

When the mouse is released, tell the selected circle that it's no longer selected, and stop keeping track of any objects for that matter.

```
function on.mouseMove(x,y)
    if TrackedObject ~= nil then
        TrackedObject.x = x
        TrackedObject.y = y
        platform.window:invalidate()
    end
end
```

When there's a tracked object, ie. a selected circle/counter, change its coordinates to match those of the mouse pointer.

```
function on.tabKey()
    if z == 0 then
        z = 1
    else
        z = 0
    end
    platform.window:invalidate()
end
```

When the TAB key gets pressed, toggle the value of *z* between 0 and 1. As you'll see below, a value of 0 means not to show the total value of the counters, and a value of 1 means to show it.
I've done a lot of hiding/showing in TNS documents before using a minimized slider. Oh how sweet it is to be able to use a simple keypress instead!

```
function on.charIn(char)
    if char=="r" or char=="R" then
        reset()
    end
    platform.window:invalidate()
end
```

on.charIn(char) gets triggered whenever a letter, digit, or other character is typed.
Once again, a very sweet UI thing to be able to use a simple keypress to trigger a function (as opposed to a minimized slider click in a TNS). Pressing "R" or "r" triggers the *reset* program.

```
function on.paint(gc)
    gc:setColorRGB(204,204,204)
    gc:fillRect(0,0,xmat,H)
    gc:setColorRGB(0,0,0)
    gc:setPen("thick","smooth")
    gc:drawRect(xmat,0,0,H)
```

Finally we get to the *paint* function which we use to draw the objects and text on the screen.
The mat gets created as a grey rectangle with a thick border (also done using a rectangle) on the right. Both *fillRect* and *drawRect* have parameters (*x, y, width, height*).

```
    for hippo, zebra in ipairs(Objects) do
        zebra:paint(gc)
    end
```

This is a different way of setting up the *for* loop. In the context of how things are defined in this activity, it behaves the same as the loops above. Obviously the names *hippo* and *zebra* are dummy names.
Basically *ipairs* is used to iterate through the elements in a table, but in an indexed way (numbered order).

```
    count()
```

Execute the *count* function each time the screen gets re-drawn.

```
    gc:setColorRGB(0,0,0)
    gc:setFont("sansserif","r",10)
    if #Objects > 2 then
       gc:drawString("(R)eset",5,0.99*H)
    end
```

Only show the "(R)eset" prompt once one has grabbed a counter. Text gets painted to the screen using *gc:drawString("text", x, y)*. It uses a pre-defined colour, and its font is determined by *gc:setFont(family, style, size)*, where style can be "r" for regular, "b" for bold, "i" for italics, or "bi" for bold italics.

```
    if z == 1 then
        gc:drawString("Press TAB to hide total value.",10,20)
        gc:setFont("sansserif", "b", 10)
        gc:drawString("Value = "..value,xmat+W/50,20)
    else
        gc:drawString("Press TAB to show total value.",10,20)
    end

    gc:setFont("sansserif", "b", 16)
    gc:drawString("+",xstart-W/50,ystartyellow-W/20)
    gc:drawString("-",xstart-W/50,ystartred-W/20)
end
```

Remember from above that *z* was used to toggle whether the value shows or not. Here's where we see that controlled.
The ". ." coding appends strings together.

Finally, add the "+" and "–" signs.

The above was meant to be a *simple* script to highlight the power of classes and tables. As you can see, it doesn't seem all that simple, but there is a lot of room for more complex elements to be added. For example:

- Add zero pairs.
- Make counters vanish when a negative gets placed on top of a positive, and vice versa.
- Use the menu to change the colours for the counters.
- Snap counters to 'grid' points.
- Use arrow keys to undo / redo.
- Give a number line representation of what's being modeled.
    - Perhaps it would be better just to split the page and use a TI-Nspire app to do that.

Other ideas?