Scripting Tutorial - Lesson 10: Taking Shape Numbers Further

Download supporting files for this tutorial

Texas Instruments TI-Nspire Scripting Support Page

In lesson 9 we created a workable document for visualizing shape numbers. Here we develop this document further using more of the techniques we have learned. We will increase the choice of display options, improve control over the document, and enhance the power of the teaching tool by allowing students to control the building up of each number for the different shapes.

Suppose instead of building up our shape patterns using grid lines, we wished to use circles? Have a think about the changes we could make to achieve this. In the sample document attached, we use a slider (view) to switch between different views, and this is certainly not too difficult to achieve (and will be left to you as an exercise in calling variables and using If conditions). Here, for simplicity's sake, we will replace the grid with circles.

Using circles to create our shapes requires only relatively small changes to our script. In particular, one approach might be to duplicate our **drawArray** function - call the copy **drawCircles**), and replace the various **drawLine** commands with **drawArc**. In fact, in most respects, it is easier to create our pattern using circles than with gridlines. One approach might be to copy and paste the For...End loop that creates each of the three shapes, then replace the multiple **drawLine** commands with something like

```
function drawCircles(number, x, y, length, height,
gc)

    types = (var.recall("type") or 1)

    if types == 1 then

    for k = 0, number - 1 do

        for m = 0, number - 1 do

            gc:fillArc(x +
            (length)*(m),

            y + (height)*(k),

            length, height, 0,
            360)

        end

    end

    elseif types == 2 then

        for k = 0, number - 1 do
```

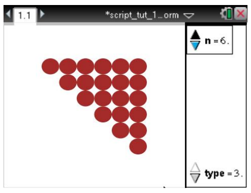
those displayed here.

You may notice that both **k** and **m** loops perform one less cycle than in the gridline example – can you see why this should be so? Study the code and try to understand it. Run it and then make changes to observe their effects.

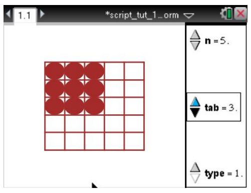
I would even suggest replacing **drawArc** with **fillArc** to give a more striking effect.

Finally, add an additional line to the **on.paint** function to the effect:

drawCircles(num, x, y, xval, yval, gc)



This could replace the **drawArray** call, or add to it (placing circles inside the grid squares).



```
            for m = 0, number do

                gc:fillArc(x + (length)*(m),

                y + (height)*(k),

                length, height, 0, 360)

            end

        end

    else

        for k = 0, number - 1 do

            for m = k, number - 1 do

                gc:fillArc(x + (length)*(m),

                y + (height)*(k),

                length, height, 0, 360)

            end

        end

    end

end
```

Lesson 10.2: Building a Dynamic Display

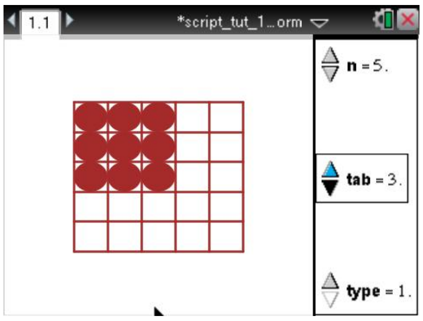
It is now a very simple matter to vary our script a little and to use the filled circles within the grid to actually show students how each pattern builds, from

```
function on.paint(gc)

    w = platform.window:width()
```

term to term. At present, the circles are filled in up to the value of **n**, just as the grid is. But suppose we introduce a new variable called, say, **tab**, which controls the number of steps displayed by our circles?

Using exactly the same **on.paint** as we did in lesson 9, we add two additional lines – shown here in **boldface**. The first picks up the current value of **tab** and stores it as a dummy variable, **tabs**. This then replaces the value of **n** as the input to our **drawCircles** function, and suddenly, as we vary **tab**, our pattern builds, step by step!



```
h =
platform.window:height()

num = (var.recall("n") or 1)

tabs = (var.recall("tab")
or 1)

xval =
math.floor(w/(num+4))

yval =
math.floor(h/(num+4))

x = w/2 - (num)*(xval)/2

y = h/2 - (num)*(yval)/2


gc:setPen("thin", "smooth")

gc:setColorRGB(165,42,42)

drawArray(num, x, y, xval,
yval, gc)

drawCircles(tabs, x, y,
xval, yval, gc)

end
```

Lesson 10.3: Finishing Up

All that remains now is fairly cosmetic. Adding arrow controls will make the document much easier to use on computer and handheld, with no need to interact at all with the sliders. In fact, if it is not desired to use this document with the TI-Nspire Document Player, the window with the sliders can actually be removed, after this step.

I chose to use up and down arrows to control the value of **n**, left and right arrows to move between the three **types**, and **tab** (as hinted) to increase the value of circle building demonstration. I also set up the **esc** key to decrease this value of **tab**.

You should once again study the sample script shown and make sure it makes sense. In most of these functions, it is wise to restrict the possible values that the variable can take. For example, pressing the down

```
function on.arrowDown()

    num = (var.recall("n") or
    1)

    if num > 0 then

        var.store("n",
        num - 1)

            else

        var.store("n", 1)

    end

end

function on.arrowUp()

    num = (var.recall("n") or
    1)

    var.store("n", num + 1)

end

function on.arrowLeft()

    types = (var.recall("type")
    or 1)
```

arrow repeatedly will lead to values of 0 and less. See how this is dealt with.

Study the **tab** and scripts closely. See how they work.

Finally, there is a missing ingredient here if we want our script to run effectively on the handheld, and not just on the computer. The handheld requires the window to be refreshed frequently, in order to paint any changes onto the screen. This may be done in a variety of ways, but one effective method involves beginning your script with the following two functions:

```
function on.timer()

        platform.window:invalidate()

end



function on.create()

        timer.start(1/5)

end
```

What these two functions do is to force the screen to repaint five times every second, thus capturing pretty much any change that is likely to be made, on handheld or computer. It is a worthwhile safety inclusion for any script you do where there are changes being made to the display.

You may also wish to add some dynamic text to raise the cognitive stakes for the user. This is covered in the example files, but you should be well able to produce such text at this point.

And with that, we conclude our introductory Lua tutorial series. Hopefully you will continue to explore and extend your knowledge and your ability to apply these techniques to your own documents. Lua offers much, much more than what we have covered, and you are encouraged to pursue further what we began here. If you want to proceed a little further along this path and learn how to use mouse controls for your Lua document, then feel free to move on to the (slightly more) advanced sequence of lessons, that begin with Lesson 11.

```
        if types > 1 then

                var.store("type",
                types - 1)

                        else

                var.store("type",
                1)

        end

end

function on.arrowRight()

        types = (var.recall("type")
        or 1)

        if types < num then

                var.store("type",
                types - 1)

                        else

                var.store("type",
                1)

        end

end
```

```
function on.escapeKey()

        tabs = (var.recall("tab") or
        1)

        if tabs > 0 then

                var.store("tab",
                tabs - 1)

                        else

                var.store("tab",
                1)

        end

end

function on.tabKey()

        tabs = (var.recall("tab") or
        1)

        if tabs < num + 1 then

                var.store("tab",
                tabs - 1)

                        else

                var.store("tab",
                1)

        end

end
```
