Making a Circular Bar in CircuitPython with VectorIO natives.

Digital Maker (DM cic) have been using a Raspberry Pi Pico & CircuitPython as the base for an explorative project, to see where it takes them (Martin & Phil).

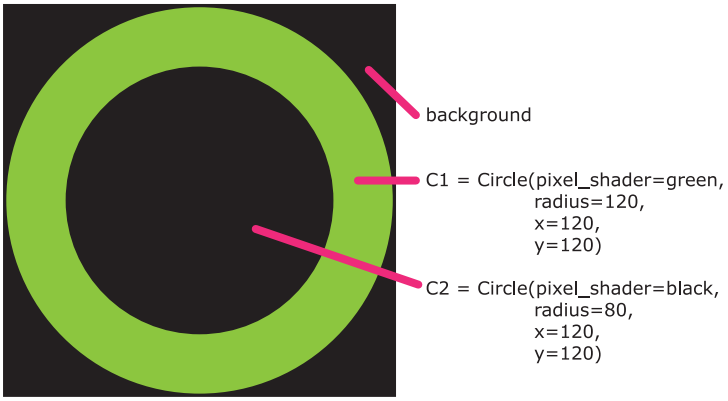
So far, DM cic have wired up an OLEd screen (240 x 240 px), two buttons, a speaker, and now, microphones (analogue first, then a PDM mic).

Phil was interested in "visualising" the inputs (having just completed making the DinkyOSC to use in conjunction with Sam Aaron's Sonic PI). A 'simple' representation of the amplitude coming from the mic was the first goal. It was fairly easy to take the minimum & maximum of the mic & convert it to a variable length "bar" or variable radius circle visualisation.

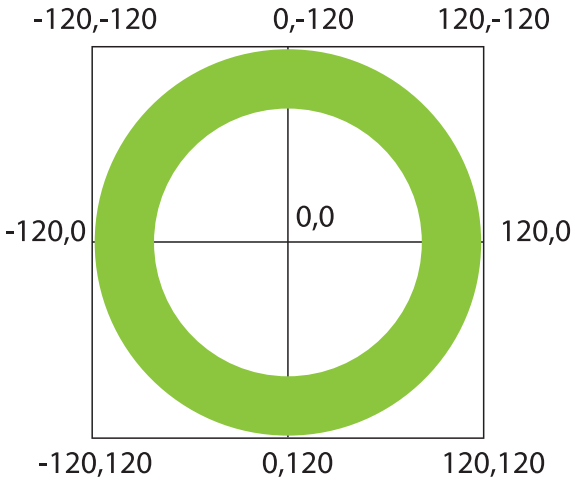
…Using a mapping tool :

```
def mapFromTo(v, oldMin, oldMax, newMin, newMax):
    # v = value passed
    # oldMin / oldMax (the min max range the V should be "inside")
    # newMin / newMax (the min max range we want the new value between)
    newV = (v-oldMin)/(oldMax-oldMin)*(newMax-newMin)+newMin
    return newV
```

Phil then thought a "circular bar" would be an exciting challenge, especially using native shapes in CircuitPython's "**vectorio**". (Phil originally looked at the displayIO.circle / rect / line etc but lacked the ability to change crucial "variable" properties quickly / on the fly). So, the limited range of **vectorio** shapes (circle, rectangle, polygon) were what he had to work with. Working on a black background, Phil created a circular bar out of two circle (one coloured circle + a black inner circle, making what looks like a thick line circle) as the base: (2nd diagram shows "fixed points" we will work with.



diag_01 (bg / circe / mask circle)

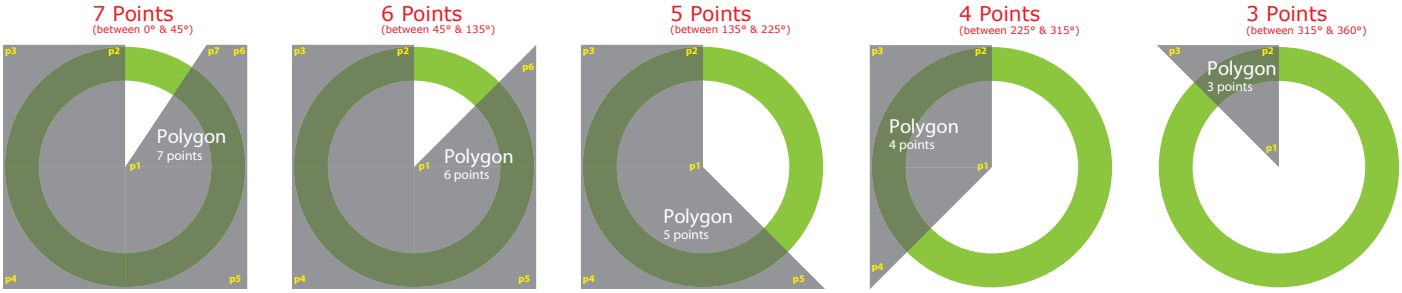


diag_02 (coordinates from centre)

Phil then thought to add a polygon to mask the visible circle, with a polygon that calculated how many points it needed "on the fly" (depending on the angle generated from the mapped value!)

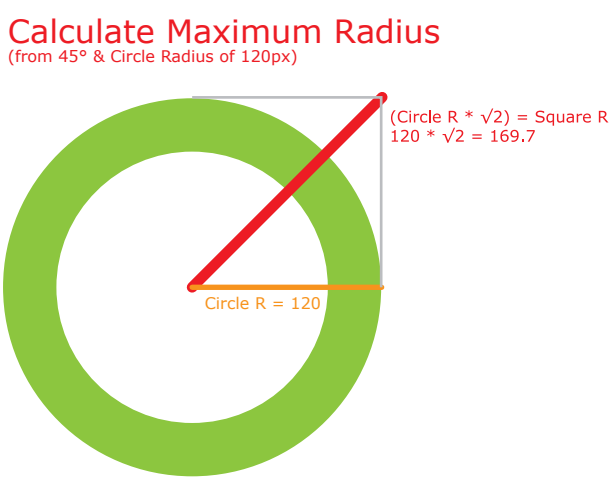
First, Phil sketched out where the points would be needed on a polygon to account for the different angles (going on 0° being the "top" of the circle, 90° (right side), 180° (bottom), 270° (left)).

From this, we can see we need between 3 & 7 points in a polygon "mask"



diag_03 (potential polygon shapes (7 down to 3 points))

To draw the polygon in **vectorio** he set some arrays with the "static points" and then later calculated the final variable point with some trigonometry. The mask should cover the circle, so the radius to calculate the variable mask point, should be "beyond" the edge of the circle (with a radius of 120px). If we take the extreme radius as the "corner" (of the square area the circle sits in), we can calculate that the hypotenuse for that by knowing our "adjacent / opposite" sides are the circle's radius (120) - and the angle is 45° - the elegant formulae is **hypotenuse = circleRadus (120) * √2 = 169.7**



diag_04 (calculating the "maximum radius" for a polygon point)

We can again use trigonometry to now work out an x & y position of the "moving point" for the polygon and add it to the "static points" stored in our handy arrays.

To do this, we can use the "extreme radius" + angle (converted from mic value). We also need to convert everything into an int (integer (a whole number)) as we can't draw fractions of pixels!

xPos = int(math.**cos**(degree * (math.pi / 180)) * extremeRadius)

yPos = int(math.**sin**(degree * (math.pi / 180)) * extremeRadius)

The new X&Y positions are "relative to 0" so if our circle is at 120,120 (the middle of the screen) we should add the new X&Y to the "origin" e.g. (originX, originY) + (newX, newP) or neatly with tuple concatenation: tuple(map(sum,zip(origin,newXY)))

Before we can add the new "variable point" we also need to figure out how many points we need in this Polygon! So, there is an "if statement" we need to run through to check where the "degree" is & return the relevant points list (starting from the centre of the circle). The great thing about **vectorio** polygons is that the x/y position sets the 0,0 "origin"... so we need to move into negative & positive values for the points. ("up" is -y , "down" is + y, "left" is -x and "right" is +x) (see Diag 02)

From diagram 03, we can see that there are 5 potential shapes, they change when the rotation passes the 45 degree points after 0°, 90°, 180° & 270°. (45°, 135°, 225°, 315°) . There is a lovely way of checking if a number is "inside" a range in python, we use the minimum, current value and the

maximum and use this syntax:

if minimum <= value <= maximum:
#Conditions are true, so do something!

Logically, this is asking "is the minimum less than or equal to the value and the value is less than or equal to the maximum". So, we can apply this to our exploration of the angle value & compile the list of points needed for the polygon mask.

```
sixpoints = [(0,0), (0, -120), (-120, -120), (-120, 120), (120, 120), (120, -120)]
fivepoints = [(0,0), (0, -120), (-120, -120), (-120, 120), (120, 120)]
fourpoints = [(0,0), (0, -120), (-120, -120), (-120, 120)]
threepoints = [(0,0), (0, -120), (-120, -120)]
twopoints = [(0,0), (0, -120)]

def calcXY(a, r=167):
    xPos = int(math.cos(a * (math.pi / 180)) * r)  # integer of new X position
    yPos = int(math.sin(a * (math.pi / 180)) * r)  # integer of new X position
    return((xPos,yPos))


def checkAngle(a, newPoint):
    polypoints = []
    if 0 <= a <= 45:  # 7 point mask top right
        polypoints = sixpoints + [newPoint]
    if 45 <= a <= 135:  # 6 point mask top right
        polypoints = fivepoints + [newPoint]
    if 135 <= a <= 225:  # 5 point mask top right
        polypoints = fourpoints + [newPoint]
    if 225 <= a <= 315:  # 4 point mask top right
        polypoints = threepoints + [newPoint]
    if 315 <= a <= 360:
        polypoints = twopoints + [newPoint]
    return(polypoints)

ang = 135
pp = checkAngle(ang, calcXY(ang))
print("polypoints = {} ({} points)".format(pp,len(pp)))
```

diag_05 (Python functions to calculate a new (x,y) location from a radius & angle and what points array needed for the polygon)

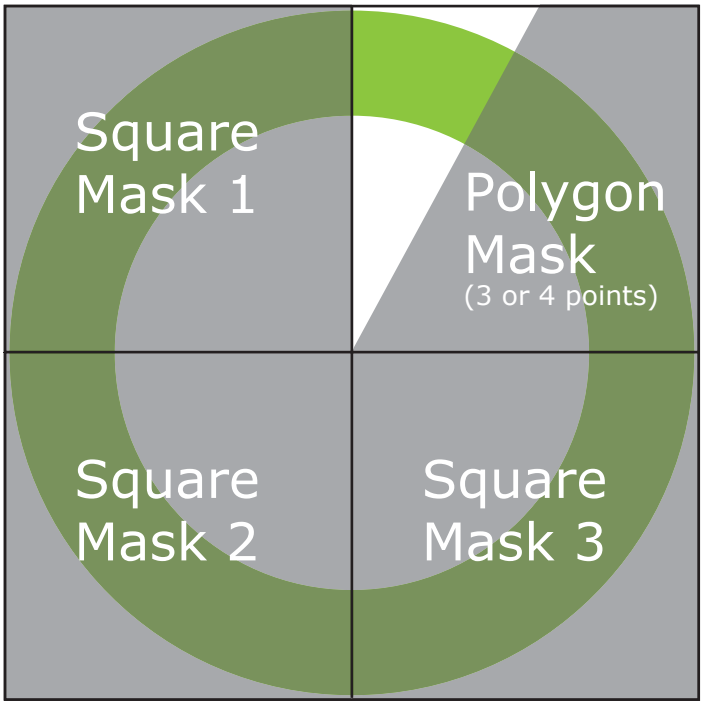
Now that we can create a polygon's points "on the fly" we can then change the **vectorio** polygon "points" parameter & the shape automatically re-draws (unlike displayio.polygon where you'd have to delete the shape, re-make the shape with the new points & re-attach the polygon to the screen objects to draw!)

polyMask = Polygon(pixel_shader=black, points=checkAngle(angle), x=0, y=0)

Once the polygon mask has been created, we can update the points with : polyMask.points = **checkAngle(angle)**(this is the beauty of using **vectorio** shapes!)

When Phil tested this, it worked, and did what was expected (!) but it was pretty slow. The "complexity" was too much for the polygon to draw a full screen "mask" (0-45 degrees, 7 points really did slow down, and with testing, phil could see the 0-135 degree polygons were a lot slower to calculate & draw than the other possible shapes).

So! Phil had one of those classic “light bulb moments” at night, just before dropping off to sleep... “Why don’t I use simpler shapes and calculate the smaller ‘odd’ polygon!” He thought. Phil decided to cerate a possible “masks” made of 3 squares (one for each corner from 90 - 360 degrees), and a small 3 or 4 pointed polygon to fill the “corner” where the variable point is... like so (Diag 06):



diag_06 (3 Square masks + one 3 or 4 pointed Polygon)

And we now only have to calculate & draw a small “variable” area + up to 3 quick squares Again, we can use an if statement to see if the polygon needs 3 or 4 points, and then draw 1, 2 or 3 squares in the areas “fully masked”.

Phil created the “CircleBar” as a python Object (OOP) - so all the calculations are contained in the Class , and you can create circular bars by making an object like this:

c1 = CircleBar(50, 1.1, 50, 50, rgb2hex(255, 0, 0), screen)

(Passing: radius, innerRadiusFactor, x position, y position, colour & the global screen to draw to)

if you visit : **bit.ly/3D6u59f** you can download the example in a zip file of the library & code.py which creates 4 circleBar objects & a counter to generate an angle to change the objects appearance.

Enjoy! If you do find a use for this, let us know, if you can help hone the code, please do!

Thanks for Reading,

yours, Phil Thompson (Digital Maker CIC lead tutor)