Cairo Graphics for tiny-C

Tom Gibson

January 1, 2019

# The python graphics library

A small library, pyGraphicsLib.tc, offers draw functions for coding in tiny-c an image with various lines, colors, geometric objects. This program, testCairo.tc, is a simple example. It makes a nonsense drawing, but illustrates its usage:

#include pyGraphicsLib.tc

#include pps/library.tc

main [

start "test\_draw", 600, 600

rectangle 150,30,240,50

moveto 100, 100

lineto 500, 500

next

ellipse 200,250,200,100

next

arc 300, 300, 150, 90, 180

next

arcneg 300, 300, 150, 0, 270

next

arc 450, 150, 30, 0, 360

dot 100, 120

moveto 100, 150

setfontsize 15

showtext "see the dot"

moveto 95, 145

lineto 80, 145

lineto 95, 126

lineto 90, 126

moveto 95, 126

lineto 95, 131

show

]

#### Code Walkthrough: testCairo.tc

Lines 1,2 load the needed libraries. main calls start, required. Arguments are the drawing name and the size of the draw window: width, height.

Rectangle draws a rectangle from 150,30 (upper left corner) with width 240, height 50. These numbers define location and size in pixels. 0,0 is the upper left corner of the window, the origin. 150,30 means go right 150 pixels from the origin, then down 30. That is the upper left corner of the rectangle.

moveto 100,100 draws nothing, but positions where to start the next draw. lineto draws from there to 500,500.

Think of moveto as lifting the draw pen, moving it over but not down. The next draw function sets the pen down and draws the defined shape. And the pen stays down. Another call to lineto and it draws a connected line. And another and another if you code that. Even a draw function like ellipse leaves the pen DOWN in its last position. And that brings us to 'next', line 17, which lifts the pen.

ellipse 200,250,200,100 draws an ellipse. The data define its bounding rectangle (not drawn obviously) and the ellipse touches all edges of that invisible rectangle.

Let's go back to the 'next' just before the ellipse call. Comment that next out and run the program again and what happens? The pen is not lifted after reaching 500,500. So, being down, it draws an unwanted line from there to the start point of the ellipse. What is the difference between next and moveto? They both lift the pen, but next does not change the current pen position. moveto does.

Arc draws a portion of a circle, or a whole circle. The first two arguments are the center point (not drawn). Then the radius, then the begin and end of the arc. These last two are expressed in degrees clockwise from the rightmost point of the circle. For example the arc in our example draws from 90 to 180 degrees. Think east/south/west/north/east (ESWNE) as the full circle from 0 to 360. You get extra points if you can pronounce the acronym!

A full small circle is drawn by arc 450, 150, 30, 0, 360.

And then a tiny dot. Look for it, its in the drawing, just above the text. setfontsize does just that, and showtext puts the text at the current location. The default fontsize is 10 which is pretty small. moveto and a pair of lineto's draw a connected pair of lines pointing towards the dot. The next three lines draw the arrowhead.

show makes it all visible, creating the window with the drawing fully done. show blocks until the drawing is brought down by clicking the frames close button (the little x), or typing alt-F4.

#### The tiny-c 1977 logo drawing

A much more elaborate drawing is logo.tc...

$ ./tc logo.tc

The code is in the file logo.tc posted as part of the github download. It uses the functions described above, plus a few new ones. arcneg draws an arc counter-clockwise, and raster is a rectangular array of dots.

It took me a whole day to do just the t. But that included my learning curve, and experimenting with techniques. Connecting lines and arcs is the hard part. The next day I did all the rest. I actually got pretty good at it after my learning curve.

Notice how I accomplished relocation of a letter from its 0,0 origin to its final placement. All those '+dx,dy+' marks do that. Ignore those for your first reading. Notice also how each letter is its own tc function. And an interesting feature: logo (line 77) does not draw the C. The logo\_y function does that. The y and C are glued together that way.

And notice the commented out call to raster at line 92. It helped me get all those numbers right. Uncover it and see what it does.

#### Hints on drawing with these tools

Techniques I learned on my first day of using these functions.

Mentally break the drawing into a set of elements (avatars, icons, tokens, whatever).

Where you see the opportunity break an element into sub-elements. Apply this recursively. Try to get it down to very simple pieces to be glued together later.

Use functions for each element. Draw the simplest first. Choose from the bottom of your recursive tree.

Use 0,0 as origin composing each element. Skip the dx,dy stuff until later. Positioning will be mechanically added later.

Do the infrastructure first in main. See the main in logo.tc for an example.

As you add a few elements to a function, save and run and check it draws what you intend. Go slow. Check often.

When a few elements work as intended, glue them together. Write a "gather together" function that calls the element functions. Add positioning so the elements don't draw on top of each other at 0,0. Here are the steps...

* With a few elements done, bring them into your editor. Add two int arguments, dx,dy.
* Notice the library draw functions have position args, the position of a lineto, moveto, circle's center, etc. Usually they are the first two arguments: lineto x,y for example. But rectangle and ellipse each have two x,y pairs. The second pair is size, not positioning. Use your editors replace command and set up to find and replace commas to +dx,dy+. Using the find and replace buttons carefully replace only those commas separating the positioning arguments. Notice you are not really replacing the comma. It is still there but surrounded by the +dx and dy+.
* Write a new function drawing them, joining them into a higher level element with reasonable guestimates for position. Then look at the result and change these guesses until things look right.
* A powerful tool in the library is the raster(width,height,delta) function. Call that in your infrastructure main just after start. The array of dots helps you get the numbers right. Comment out the raster to see the pure drawing so far, and uncomment it to position the next item.

The hardest thing is blending an arc into a line smoothly. What makes arc (and arcneg) troublesome is the end points are a mystery. Position is done via the 5 parameters, and the end points need to touch the adjacent line. And often they must touch smoothly, no angle. Here are some tricks that help...

* Draw the two lines that need to be joined with an arc
* Temporarily use moveto between them:

lineto 10,20

moveto 20,20

lineto 10,30

* display that over a raster
* guestimate the center point of the arc. A small error doesn't matter.
* guestimate the radius and two angles.
* type the arc with this guestimate AFTER the first lineto and BEFORE the moveto.

lineto 10,20

arc 15,25,5,135,45

moveto 20,20

lineto 10,30

* if its crazy try argneg. Remember arc draws clockwise, argneg counter-clockwise. Your arc direction should conform to the general direction of your line drawing.
* diagnose the issue: OPPS that 135 should be 225.
* the little connecting line from 10,20 to the beginning of your arc is your friend. Jiggle the arc's center to make that line tiny, eventually zero. Add radius and angles to the jiggling until it connects the original two lines.

As you do more of these connections you will quickly diagnose which parameter to jiggle.

IF SOMEONES WILLING to do the math, I would love a new draw function called bow. Its really an arc with different data: the two end points plus a dimension from the arc to its chord in the middle.

# How it all works, quick overview

There are two executables involved. One is the tiny-c interpreter, the other is a python script, cairopy.py. It all starts in tiny C. Write a simple draw program, such as testCairo.tc, and run it...

./tc testCairo.tc (linux)

tc.exe testCairo.tc (windows)

testCairo.tc writes a file, testCairo.draw, capturing the draw commands. Then it uses the library function 'system' to start cairopy.py, which reads the file and renders the drawing in a new viewing window. During drawing the system function is in a wait state. During viewing both executables are in a wait state. Click the little X button in the viewing window and the window goes away, cairoGraphics wakes up and exits, and tiny-c resumes. But testCairo.tc is done, so tiny-c exits.

Closing the viewing window does not erase either the draw file or the png of the drawing, testCairo.png.

The python script can be run standalone with the draw file as the argument...

$ python cairopy.py testCairo.draw

The file test\_draw.draw has not been erased, and is redrawn. This demonstrates the two executables are totally independent. All tiny-c does is write commands to a file. All cairoGraphics does is read the file and render the drawing as a png, and pops that png into an image window.

In fact you can make several drawings, each with a different name and their files persist for later use.

#### Script cairopy.py

cairopy.py is a python script, part of the download. The script starts with definitions: window, draw, and show. window creates the window, its drawing surface and sets its background white and the drawing tool black. Notice the arguments to ctx.set\_source\_rgb(1,1,1). Cairo uses a floating point with range 0 to 1 for the intensity of each color. Tiny-c is an integer programming language, no floats. So it uses an integer with range 0 to 255 for these intensities. So the four lines:

ctx.rectangle(0,0,width,height)

ctx.set\_source\_rgb(1,1,1)

ctx.fill()

ctx.set\_source\_rgb(0,0,0)

are pure cairo. We will see later how tiny-c's setrgb works.

draw(x) is the magic. The draw file has command names followed by integer and/or quoted strings. x is that string parsed into a list of integers and text elements. draw(x) builds string as a python statement, and eval(string) executes it! The comment (its red in my editor) lines 12..20 “hard way...easy way” explain the magic. Since most ctx commands have only numeric or string parameters, draw implements probably 100 or more possible command lines from the passed in file. More on this issue coming up.

Finally show creates the png file, and if a show\_app is declared (default is “display”, an image magik app) it pops the png into that application.

So window does the infrastructure setup, draw called repeatedly does the drawing, and show does the infrastructure wrap-up. But what about argument variation, weird arguments, like radians? Skip down to line 51, the comment “It all starts here”, and from there to line 61 just after defining M\_PI. (I hope thats enough precision.)

We've seen the clash between tiny-c's ability to produce arguments versus cairo's definition of parameters. The int vs float is a minor issue, python (like C) promotes numeric arguments from int to float. So ctx.set\_source\_rgb(1,1,1) is equivalent to ctx.set\_source\_rgp(1.0, 1.0, 1.0). No issue there. But how to express orange, which happens to be (1, fraction, 0)? The answer is argument munging, and that is the dozen (so far) elif's in lines 64..103. The call to draw is in the else, line 105, at the very end.

And its not always arguments, we have some command name munging, too. Most notably setrgb (tc's library function) into set\_source\_rgb. And some overhead non-draw commands like showapp. Cairo knows nothing of showapp. The non-draws (window, show, showapp) are outside of cairo's domain which is drawing to a surface and writing to a png. But the python script cairopy.py needs to know these. I call them “specials”...

* argument munging
* command name munging
* non-draw commands

They are caught and handled especially and not passed to draw(x). Hopefully they are few in number and the magic of draw(x) will do most commands. Notice how arc turns degrees into radians, and setrgb the 0-255 range into 0.0-1.0 float range.

# Adding draw tools

Cairo is an incredibly rich tool set. The few tools provided by the code described above is enough for the very primitive stuff I have done so far. The path forward is to add more of these tools, fill, text, color, gradients, all that fun stuff. The motivation is the next drawing I (or you) want to do. To do my tiny-c logo example I added next and arcneg. And I added color and fill for the color.tc demo.

Here are the visit points to add a cairo function:

cairopy.py for specials

Note this is not needed if draw(x) is satisfactory. Also for my sanity I have been keeping these in alphabetical order. And at the end of the file as a comment are a couple of template lines to cpy/paste for a quick start.

* add elif and treatment, that's all.

pyGraphicsLib.tc

Study functions cs, ci, cc, nl (lines 42…) and their usage. These aids help define the command line. Each line must start with a command string, cs <name>, and end with an nl. I find these four little functions strung together on one line much easier to see they are correct. Hence the very short names.

Also notice not all draw functions build command lines directly, dot, raster, and especially ellipse for example. They just call other functions that do.

* add function after the comment: //actual draw tools
* use a simple function name, lower case for ease of typing, suggestive of its cairo equivalent, <name>. It need not match the <name> written to the file. arcneg is an example.
* choose <name> carefully: if not a special it must match the cairo name, because draw(x) will use that name.

Notice for my sanity … alphabetical …. (thank you).

# Going forward

I recently added colors, fill, and text. I have no intention of doing a full set of cairo bindings, just those I would like to use. Tiny-C is a teaching tool, not a modern production tool. I am using it as a learning tool for myself.

#### Sample Programs

Currently these three programs in the SamplePrograms directory do Cairo drawings:

testCairo.tc, logo.tc, and color.tc

#### man page

The appendix is the man page for cairoGraphicsLib.tc as of this writing.

Tom Gibson

January 1, 2019, Happy New Year

# Appendix – pyGraphicsLib.tc

Dec 30, 2018

Tom Gibson

These are the functions for this library as of this writing.

#### internal data

char \_cgdrawname(50) //users name for drawing, no spaces

char \_cgfilename(55) // file passing draw vector to cairopy.py

char \_cgcmd(70) // system command to start cairopy.py

char NL, QUOTE // defines these two characters

#### functions

Arguments are either integer or character string. All integer arguments are pixels unless otherwise noted. Recall that char xx(0) in tiny-c defines xx as a string. A quoted string is valid, and probably most common.

**start** char name(0); int window\_width, window\_height

Defines internal \_cgdrawname, \_cgfilename, \_cgcmd, newline (NL), and quote (QUOTE). Open \_cgfilename for writing. Issues window command.

**showapp** char app\_name(0)

Define the name of the program that will show the png file. The linux default is “display”. If defined to “” (zero length string) the png is produced but not displayed. This latter arrangement is useful in scripts.

**show**

Closes \_cgfilename. Invokes \_cgmnd which reads the file, creates the png file, and if a showapp is defined invokes that application to show the png file.

**arc** int centerX, centerY, radius, degreeStart, degreeEnd

Draws a circular arc, or a whole circle. Arcs are drawn clockwise from the point determined by the center, radius and degreeStart, to the point determined by degreeEnd. Degree==0 is East. Note that like **lineto** the pen if down will draw from its current location to the start point. And after the arc is drawn its location is the end point.

**arcneg** int centerX, centerY, radius, degreeStart, degreeEnd

Like **arc** except counterclockwise

**dot** int x,y

Draws a dot, actually a line one pixel long.

**ellipse** int upperLeftX, upperLeftY, width, height

Issues a series of commands that results in drawing an ellipse.

**fill**

Fills closed lines with the current color

**lineto** int x, y

Draws a straight line from the current position to x,y.

**next**

Lifts the pen.

**moveto** int x,y

Lifts the pen and sets current position to x,y.

**rectangle** int upperleftX, upperleftY, width, height

draws a rectangle

**raster** int width, height, delta

A drawing aid. From 0, 0 to width, height draws dots at delta intervals horizontally and vertically.

**setfontsize** int size

Sets the font size for subsequent **showtext**s. Default is 10.

**setrgb** int r,g,b

Sets the current color to red=r, green=g, blue=b. Arguments are in the range 0..255. Larger arguments are reduced to 255.

**showtext** char t(0)

Draws text whose baseline starts at the current position.

**stroke**

Lines are made visible in the current color.

* + - 1. **xersect** int ix(1), ax0,ay0,ax1,ay1, bx0,by0,bx1,by1

#### About fill, stroke and path

The pyGraphicsLib.tc function lineto(x,y) “draws a straight line … to x,y” according to this document. The line is invisible until stroke is called. The function maps to the Cairo function cairo\_line\_to. The cairo documentation says this using a “path” terminology: “Adds a line to the path.” Paths are always invisible. A drawing has multiple paths and the one being drawn is the “current path.” Paths have two roles: to outline a figure or to draw actual visible lines. For the former fill colors the figure. For the latter stroke make lines visible. If you draw several paths fill will color all closed paths, current and not current, and stroke will draw all lines. Once filled or stroked they are done, subsequent fill/strokes will not modify them. It is important to understand this last sentence.

One more fact: the tc function show issues a stroke just bo be sure everything is visible.

For example, the logo.tc code draws five closed paths, the y and C being combined into one, and the i sporting two. All lines, arcs, etc for all of these are done in the functions. They do not fill or stroke. At the very end one call to fill colors them all. The subsequent stroke call in show does nothing.

This is my current understanding of Cairo's behavior, and may help your transition to Cairo documentation.

Reference: <https://www.cairographics.org/manual-1.0.2/cairo-Paths.html>

-----

This is a test line to see why new text above (xersect) has weird indenting. This is a test line to see why new text above has weird indenting. Given two line segments determine their intersection point. Returns -1 of parallel lines. Returns 0 and sets ix to the intersection if lines are not parallel. Note the intersection can be outside the drawing area.

* + - 1. Given two line segments determine their intersection point. Returns -1 of parallel lines. Returns 0 and sets ix to the intersection if lines are not parallel. Note the intersection can be outside the drawing area.
      2. This is a test paragraph to see if the above problem repeats. This is a test paragraph to see if the above problem repeats. YES IT DOES.
      3. I replaced my profile, this one is running a new one, default. Libreoffice 5.0.3.2.