Introducing Programming to the Visually Impaired

One Pixel at a Time

# Introduction

I plan to provide you a demonstration of my work which aids the visually impaired in learning programming with both tactile and auditory feedback to graphical programs they write. I will follow up, covering the methods I have used. If time permits, I will include paths I might have taken in the implementation. First, I will give a short outline as to how this came about.

I’ve been a programmer for over half a century. I started programming in college. There I was going to be a Mathematician. But sadly, when I took my first non-required math course, late in my sophomore year, I thought I might be able to do this but my heart isn’t in it. I scrambled to find a major. Classical Electronics was out – to many around were much more ahead of me. Digital Electronics – a bit better – less abstract than Math but still logical. Programming – that’s it – no knowledge required here, just a game. So, a career was started. And I got paid.

Currently, I am, depending on my mood, self-employed, retired, or unemployed. During the recent few years, I’ve been volunteer teaching programming to those who have “missed the programming boat”. These people are not programmers, but are interested in finding if programming might be for them. They are curious at least at what programming can be – not just the mad typing on a console one might see on TV. My courses are not for those programmers who use programming language *X* and want to pick up the programming language Python, which I demonstrate in my teaching. There are many good in-person courses, online courses, and tutorials. My courses start by presenting the student a ready-made program starting very simple like ‘print(“Hello world!”)’. I run it for them. They, in turn modify the program, and run it again. But I found that, even simple programs were often too abstract, without much “action” or feedback. I hoped to increase the “action” to hold the fledging programmer’s interest. We moved to simple graphical program examples, e.g., display a square. To facilitate this, we use Python’s **turtle** graphics module.

Along the way, aided by the presence of the famed **Perkins School for the Blind** a few miles from my home, it hit me – these fine and powerful graphical examples were next to useless for the prospective programmer can’s see the pictures. What’s left? Should we restrict programming examples to text-based output? Possible, but not very nice. Given the scarcity of widely available economical graphical display devices for those who can’t see, we are stuck. Rather than tackle the large problem of general graphical presentation for the visually impaired, I decided, to concentrate on making simple graphical programs more visible.

My work promotes “viewing” program graphics by adding **extended graphics views** to programs containing turtle statements. An extended **graphics view** is any mechanism which provides an additional view to the traditional displayed graphical view.

# A Demonstration (Hopefully Live)

## Samle Program – square.py

# square.py

# Display a square

from turtle\_braille\_link import \* #link to lib

color(*"green"*)

width(2)

forward(200)

right(90)

forward(200)

right(90)

forward(200)

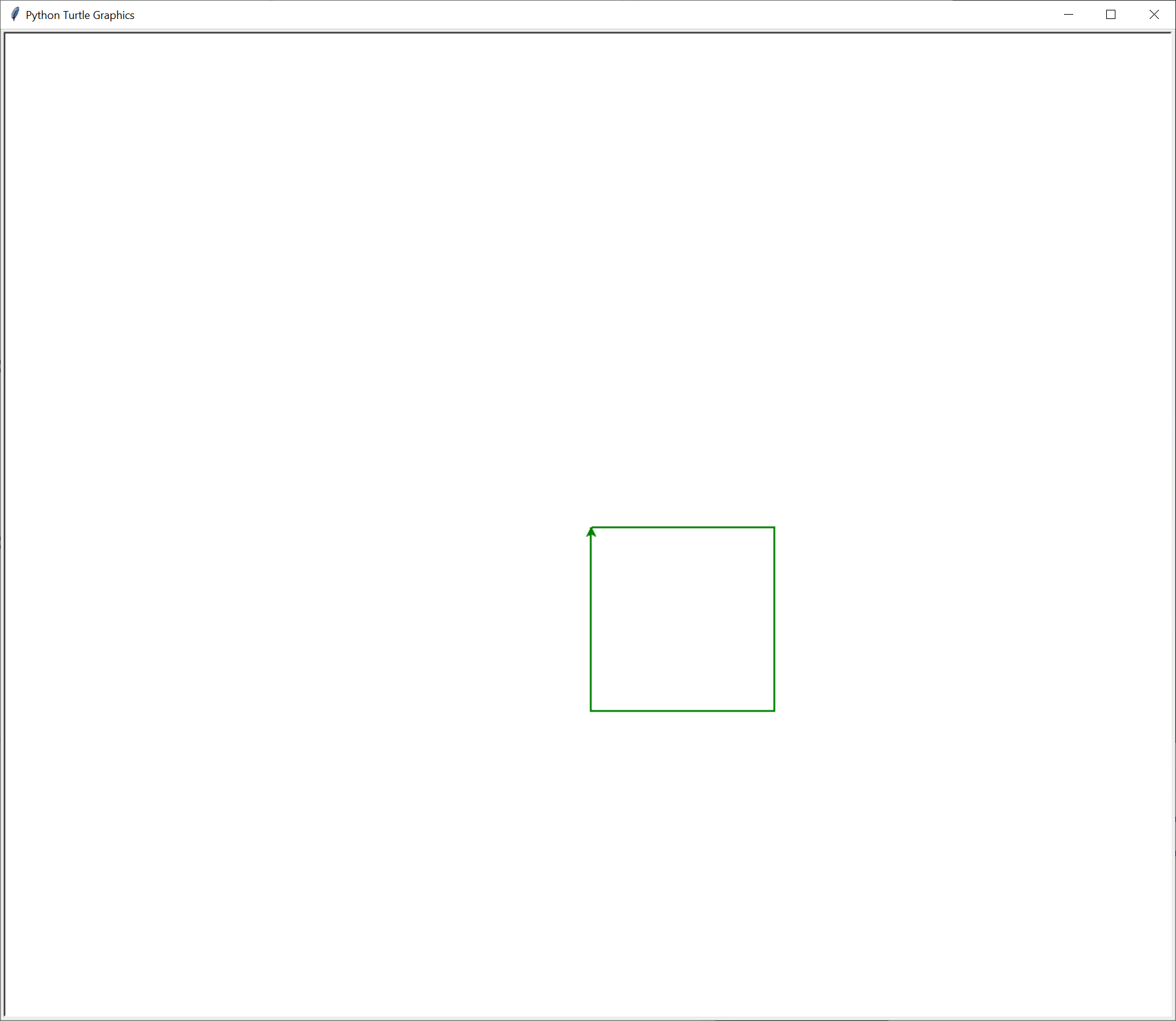
right(90)

forward(200)

done()

## Sample Program Output

### Turtle Graphics Screen



### Standard Output, Including Text for Brallier

resource\_lib\_proj/src is already in path

Creating Log File Name: C:\Users\raysm\workspace\python\resource\_lib\_proj\log\square\_20221116\_110414.sllog

**… omitted for brevity**

canvas width: 804

canvas height: 804

Lower left: min\_x:-20 min\_y:-176

Upper Right: max\_x:220 max\_y:-16

green at row18 column20

Braille Display - Braille Print Output

,,gggggggggggg

,,gg,,,,,,,,gg

,,gg,,,,,,,,gg

,,gg,,,,,,,,gg

,,gg,,,,,,,,gg

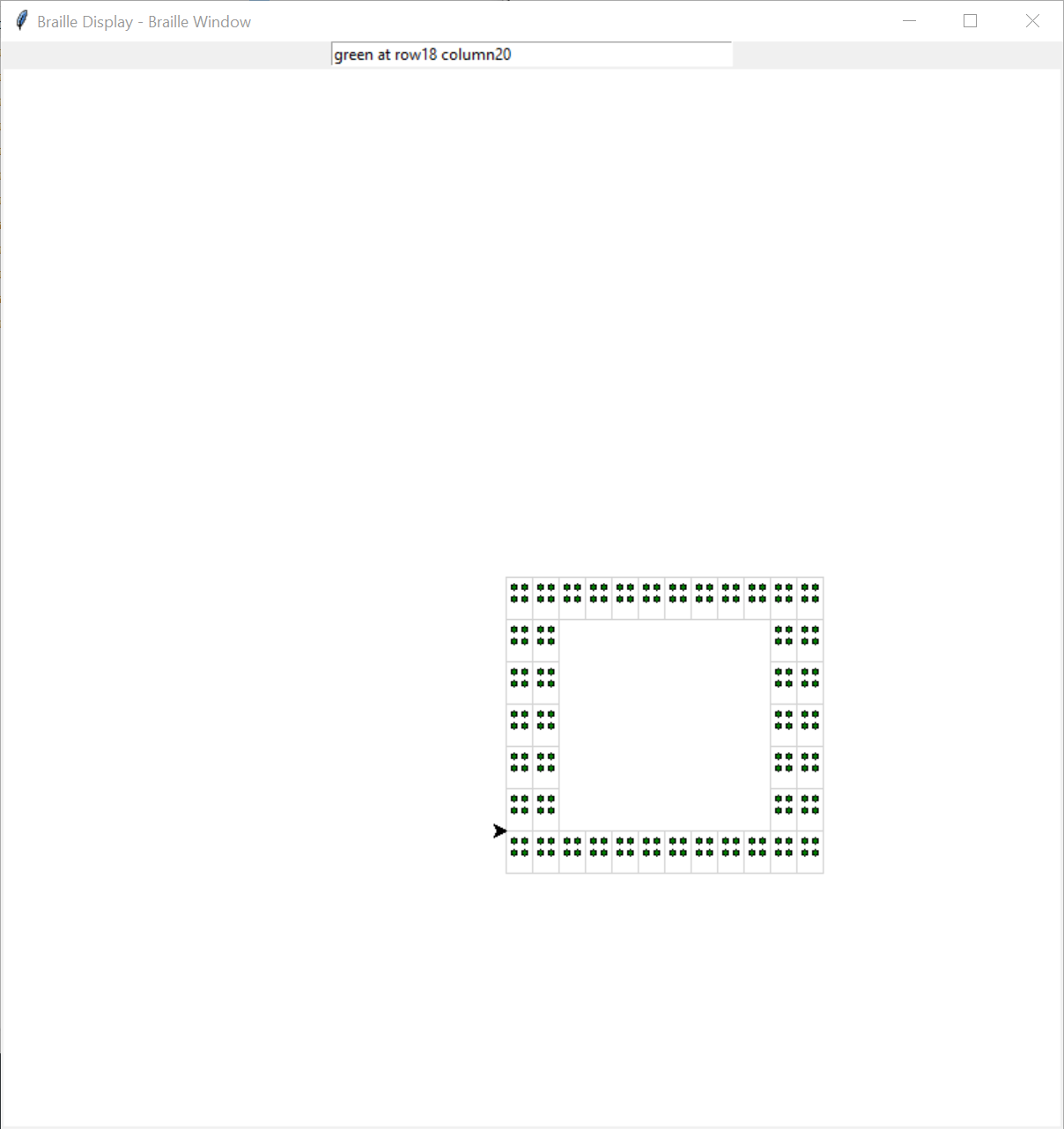
,,gg,,,,,,,,gg

,,gggggggggggg

Standard Output, Including Text for Brallier

* “g” – Color is represented by first letter of the color string, e.g., “g” for green
* “,” – Each non-trailing space is replaced by a “,” whose, braille representation empty-like (only one of the possible six dots) but prevents Braille machine software from compressing multiple-space strings.
* Figure(s) moved closer to the left and top edges to make the figures easier to find.

### Braille Window – View of Brailed Figure



* Surrounding rectangles are present to make it easier for me to see groupings.
* The dot color is just to remind me of the color represented by the Braille symbol.

### Braille Window Feedback

The braille window was first created as a development/documentation tool. Its primary function was to provide me, a sighted person with no access to a Braille machine, a way to better visualize the braille produced when the text output was passed to the actual Braille machine. I enhanced the Braille Window to help the user navigate around the window, reporting the current cursor position. The user can, via keyboard key presses and mouse operations, move to around the window. Cursor position is reported as distance from the closest figure plus the cursor position on the Braille grid (default 25 rows, 40 columns). Position is spoken and printed by default.

Help Message (keyboard commands)

*h - say this help message*

*Up - Move up one row*

*Down - Move down one row*

*Left - Move left one column*

*Right - Move right one column*

*a - Start reporting location*

*c - color/clear cell*

*l - Start logging talk*

*m - Stop logging talk*

*s - Stop speech*

*t - Start speech*

*v - make figure cells visible*

*w - make figure cells invisible*

*x - exit program*

*z - Stop reporting location*

*RETURN - Report location*

*Escape - flush pending report output*

* Keyboard commands are case insensitive
* Up, Down, Left, Right are keyboard arrow keys
* a – Start reporting location (default) adds current row, column to end of position reporting messages in the form “at row *N* column *M*”
* c – color/clear cell makes cell at current cursor position visible
* l – Start logging talk (default) – begin sending talk speech to standard output (console)
* m – Stop logging talk – Stop logging/printing speech. This may be helpful, if alternate speaking of the console, such as created by JAWS or NVDA programs is confusing the listener.
* s – Stop speech stops the talking. This may be helpful for sighted users as it greatly increases screen navigation speed.
* t – Start speech (default) starts talking reports
* v – Make figure cells visible(default) reverses w
* w – Make figure cells invisible – gives the sighted viewer a sense of the difficulty navigating when one cannot see the drawing.
* x – exit program – stops program execution
* z – Stop reporting location – removes “at row *N* column *M*” from end of position reports.
* RETURN – Report location – forces location report to remind listener/viewer.
  + Position Report Examples:
    - At top edge at row1 column15 – *Cursor is at top, at row 1 from top, column 15 from left edge*
    - down 9 at row4 column23 - *Figure is down 9 rows*
    - right 7 down 12 at row1 column13 – *Figure is right 7 columns, down 12 rows*
    - green at row13 column20 *– Cursor on green cell*
    - down 1 at row12 column20 *– Figure is down 1 row*
    - At right edge at row16 column40 – *Cursor is at right*
    - up 5 – Figure is directly up 5 rows, no “At row…column” reported
    - green – Cursor on green cell, no “At row…column” reported

# Requirements / limitations:

* Environment
  + Extra files need to be in path (GitHub: raysmity619/resource\_lib/src/\*)
  + A directory path named “resource\_lib\_proj/src” must exist or be created somewhere within the working directory path. The extra files named above must be placed in this directory (src).
  + Installing pyttsx3 is required for program generated speech. If not installed reporting is limited to console printing. (“pip install pyttsx3” works for me)
* User coding
  + Use “from turtle\_braille\_link import \*” instead of “from turtle import \*”.
* Turtle feature limitations
  + 80-90% of language – no animation support
  + Limited color support
  + circle() not well supported – no time
  + shape() plus some others – no time

# How It Works

## Overview

### Function flow

TurtleBraille and BrailleDisplay are both classes. In the following the terms refer to instances of their respective classes.

1. The turtle function **forward(100)** is called in a program.
2. External function **forward()** in turtle\_braille.py called.
3. **forward()** calls TurtleBraille’s **forward()**.
4. TurtleBraille’s **forward**() calls BrailleDisplay’s **forward**().
5. BrailleDisplay’s **forward**():
   1. Calls turtle functions **forward**() to create turtle visual effects THEN
   2. Does processing to create **extended graphics view** components. These components are usually instances of the class BrailleCell.  Each **extended graphics view** display is created by processing all the created BrailleCell instances.

### Linkage

“from turtle\_braille\_link import \*”:

* Searches for directory resource\_lib\_proj/src, if not already in search path, from the current directory up and add this to the search path.
* Does a “from turtle\_braille import \*” to add turtle function names to programs name space.

### Limitations

Not all turtle functions are implemented. For example, no animation support is available.

For object-oriented turtle operation, one must instantiate an instance of BrailleDisplay and call turtle member functions through this object. Multiple instances of turtle objects are not currently supported.

### BrailleCell – Unit of Display

Each instance of the class BrailleCell represents a small rectangle display region. These regions are members of a grid covering the turtle display. These regions will be rendered to produce the **extended graphics views**. The currently implemented **extended graphics view** displays are:

* **Text output**: rectangular array of letters (first letter of color, e.g., “r” for red, at this point) to be sent to Braille machine which produces a coarse resolution picture
  + Currently 40 columns by 25 rows
  + This display text is placed in the program’s standard output (console) when the user program calls turtle function done() / mainloop().
  + Actual transmission to Braille machine and producing of output is not currently part of the program.
  + A few transformations are made after the cell generation to facilitate Braille reading:
    - Each space of multiple non-trailing spaces is converted to a “,”
    - The figures are moved closer to the left edge and top edge.
* **Braille Window**: A window presenting a visual rendition of the Braille machine out. This window can be navigated by the user, producing spoken and/or printed feedback, indicating the current cursor position.
  + Currently 40 columns by 25 rows
  + This display window is created when the user program calls turtle function done() / mainloop().
  + Braille is simulated with rectangular cells containing colored dots to simulate actual Braille.
  + The transformations for the Braille targeted Text output are omitted

### Rectangle / Cell / BrailleCell Population and Figure Drawing

An **extended graphics view,** representing the displayed turtle graphics view**,** is composed of BrailleCell instances which represent a rectangular region. The **extended graphics view** picture is created in two steps:

1. Generate points based on an estimated sampling of the figure created to imitate the figure generated by the turtle commands which are placed in the appropriate cells (ix,iy).
2. Go through all the generated cells to create the pictures. Note that the traversal depends on the **Visually Augmented Display**:
   1. To generate the text picture one must go left to right for each line, placing spaces for missing (ix,iy) cells.
   2. Generating the Braille display does not require a special ordering because the cell drawing only depends on the (ix, iy) tuple and nonexistent cells are left undrawn.

### Filling and getting the Point

Filling regions weather they be the inside of a polygon with the “fill” property or a line of some width greater than one is central to generating our pictures. Note that for our purposes, a line is just a thin rectangle. The task is somewhat simplified for us because we need only calculate at least one point per rectangle/cell. Over sampling presents no problems. Our process has two major parts:

1. Divide the figure or sub-figure into triangles.
   1. Use an algorithm that assumes a convex polygon (TBD fix this) – start lines from a head point, creating a set of triangles from the figure.
2. Populate each triangle.

From braille\_display.py function **get\_points\_triangle** docstringcomment:

def **get\_points\_triangle**(*self*,p1,p2,p3, point\_resolution=None):

*""" Get points in triangle*

*The goal is that, when each returned point is used in generating the including cell, the resulting cells completely fill the triangle's* *region with minimum number of gaps and minimum fill outside the triangle. Strategy fill from left to right with vertical fill lines*

*separated by a pixel distance of point\_resolution which will be converted to fill points.*

*Strategy*

*\* pxs[1]*

*\* \**

*\* \**

*\* |\**

*\* | \**

*\* | | \**

*\* | | \**

*\*| | | \**

*\* | | more | \**

*\* | | lines | \**

*\* | | | | \**

*pxs[0] \* | | | | |\**

*\* | | | | |\**

*\* | | | | \**

*\* | | | | \**

*\* | | | \**

*\* | | \**

*\* | | \**

*\* | | \**

*\* | | \**

*\* | \**

*\* | |\**

*\* |\**

*\* \**

*\* pxs[2]*

*Begin by adding points directly included by the* *triangle's three edges. Then continue with* *the following.*

*Construct a series of vertical fill lines separated by point\_resolution such that the fill points from these lines will appropriately cover the triangle. … more details*

### An Example Detailed Flow – forward()

* Code file names are in bold to right.

**file\_name.py**

* Timing comments are as such:

Time: Action

* Program code snippet:

Code from file

**square.py**

1: Initial call from program

forward(200)

**turtle\_braile.py**

Earlier: before – at program beginning

tum = TurtleBraille()

2: Called by forward() in **square.py**

def **forward**(length):

return tum.forward(length)

Earlier: before - at program beginning

class TurtleBraille():

…

bd = BrailleDisplay(win\_width=self.win\_width,

win\_height=self.win\_height,

grid\_width=self.cell\_width,

grid\_height=self.cell\_height)

…

self.bd = bd

3: Called from forward() in **turtle\_braile.py**

def **forward**(*self*, length):

return *self*.bd.forward(length)

braille\_display.py

4: Called from TurtleBraille’s forward() in **turtle\_braille.py**

def forward(self, length):

""" Make step forward, updating location

"""

rt = self.tu.forward(length)

x1 = self.x

y1 = self.y

angle = self.angle

rangle = angle/180\*pi

x2 = x1 + length\*cos(rangle)

y2 = y1 + length\*sin(rangle)

self.goto(x=x2, y=y2)

return rt

**5**: Called from TurtleDisplay’s forward() in **braille\_display.py**

def **goto**(*self*, x, y=None):

rt = *self*.tu.goto(x,y)

x1 = *self*.x

y1 = *self*.y

x2 = x

if y is None:

y = *self*.y

y2 = y

*self*.add\_line(p1=(x1,y1), p2=(x2,y2))

return rt

6: Called from TurtleDisplay’s goto() in **braille\_display.py**

def **add\_line**(*self*, p1=None, p2=None, color=None, width=None):

*""" Add new line*

**:p1:** *xy pair - beginning point*

*default: previous point (add\_point or add\_line)*

**:p2:** *xy pair - ending point*

*default: previous point (add\_point or add\_line)*

**:color:** *line color*

*default: previous color ["black"]*

**:width:** *line width*

*default: previous width [1]*

*"""*

if p1 is None:

p1 = (*self*.x,*self*.y)

if p2 is None:

raise Exception(*"p2 is missing"*)

if *self*.is\_filling:

*self*.add\_to\_fill(p1,p2)

if width is None:

width = *self*.line\_width

if color is None:

color = *self*.\_color

if *self*.is\_pendown:

points = *self*.get\_drawn\_line\_points(p1, p2, width)

*self*.populate\_cells\_from\_points(points, color=color)

*self*.x, *self*.y = p2

# Admissions and Apologies

It’s time to fess-up, come clean, and so on. Here are a few, not a complete list, of things we could/will do better. Note the programming here is mine. For “we”, where it is admirable substitute myself plus folks who have taught or advised me, where it is a mistake or disaster substitute “me”.

## OK – I’ts not done

Most of my recent works are mostly prototypes. The rest are strictly prototypes. My greatest goals have been to investigate how my programming could make something interesting, fun, and maybe useful. Almost Fifty years ago Fred Brooks published a great software engineering book *The Mythical Man-Month: Essays on Software Engineering*. In this book, referring to projects much, much grander than mine, he advised one to “Build two systems and throw the first away”. In which he asserted that, in the first, we could learn the “big” lessons, and in the second we could “make it right”. Well, I’m mostly trying to learn the ”big” lessons. Possibly, for those short on time, we might follow the abridged plan – “Build one system and through the first away”. Of course, Mr. Brooks would strongly agree that this does not absolve us, the designers, implementers, deliverers, from using all our efforts in doing the best we can on the first **and** subsequent versions. I’m doing the best that I can. *It’s hard to be humble*…

## Incomplete Turtle Language Support

As we mentioned, no animation support is provided. I couldn’t see how picture movement could be delivered to the targeted audience – the visually impaired. As of the addition of audio feed back, I’m reconsidering that issue. Some things, like circle(), shape(), begin\_poly(), and a host of others – just didn’t have the time. Please remember the primary goal – build a system to support the writing of simple programs.

## Little or no Testing

Outside a few instances, testing, really just exercising, is relegated to the self-test code at the end of the major class files. As this is a heavily interactive program, it’s in major need of testing.

## The Fill Programming Needs Work

The lackluster performance with poly squares with thin edges, could be improved. Filling of non-convex polygons needs work or at least testing.

# Roads Not Taken

Here we look at least one alternate development path.

## Turtle direct, then “Scraping Canvas Items”

Python’s turtle uses the tkinter module for graphics. Virtually all graphical elements become tkinter Canvas items. One possibility for our graphics would be to send all turtle operations directly to turtle, as we do now, but then, waiting till completion (done(), mainloop() call) check the canvas items, constructing the BrailleCell components from those items. This approach has several benefits:

* Little or no need to implement turtle shadow functions to construct BrailleCells such line, fill, …
* The “Scraping Canvas” operation would be directly applicable to tkinter programs too.

Why didn’t we try it? Well, we did. See our GitHub.com/raysmith619/TurtleBraille/README.md for details. The bottom line, so far, is we almost got it to work, but we could not solve the display problem. We go it to display ALMOST correctly. The most embarrassing thing is we could no figure the canvas item coordinates quite right. For some reason we kept seeing negative tinker coordinates which should be non-negative. The sad result was the closest we got were almost perfect figures but image reversed.

However, we will outline a bit of the strategy used. We still might go back.

The basis for generating cells from the tinker canvas is demonstrated in the following functions from the file braille\_display\_2.py

The BrailleDisplay class member function **populate\_cells\_from\_canvas()** “scrapes” the tkinter Canvas instance, collecting appropriate items, created by turtle and creates representative cells for an extended graphical view. The member function **find\_overlapping()** does the “heavy lifting” by determination of “appropriateness”. The crucial tkinter function is **find\_overlapping**() which returns a list of canvas items for those items that overlap a given rectangular area.

|  |
| --- |
|  |

# Some levels are shifted left for presentation

class **BrailleDisplay**:

*… more member functions*

def **populate\_cells\_from\_canvas**(*self*):

*""" populate cells covered by canvas objects*

*"""*

canvas = *self*.rt\_canvas

for ix in range(*self*.grid\_width):

for iy in range(*self*.grid\_height):

cx1,cy1,cx2,cy2 = *self*.get\_cell\_ullr\_win(ix=ix, iy=iy)

items\_over = *self*.find\_overlapping(canvas, cx1,cy1,cx2,cy2)

# ... debugging

if len(items\_over) > 0:

top\_item = items\_over[-1]

canvas\_id = top\_item

# ... more checking/debugging

point\_win = (cx1+cx2)/2,(cy1+cy2)/2

point\_tur = *self*.cvt\_win\_pt\_to\_tur(point\_win)

# ... lots of debugging

*self*.update\_cell(pt=point\_tur, color=color,

canvas\_id=canvas\_id)

def **find\_overlapping**(*self*, canvas, cx1,cy1,cx2,cy2,

include\_annotations=False):

*""" Get canvas items (Canvas.get\_overlapping)*

**:canvas:** *Canvas object - None ==> use self.rt\_canvas*

**:cx1***,cy1: rectangle upper left corner*

**:cx2***,cy2: rectangle lower right corner*

**:include\_annotations:** *include annotations*

*default: False - ignore annotations*

**:returns:** *list of canvas items overlapping rectangle*

*"""*

if canvas is None:

canvas = *self*.rt\_canvas

items\_over\_raw = canvas.find\_overlapping(cx1,cy1,cx2,cy2)

if include\_annotations:

items\_over = items\_over\_raw

else:

items\_over = []

annotated\_items = canvas.find\_withtag(*self*.annotate\_tag)

annotated\_set = set(annotated\_items)

for item in items\_over\_raw:

if item in annotated\_set:

is\_annotated = True

else:

is\_annotated = False

SlTrace.lg(*f"overlapping:{item} annotated:{is\_annotated}"*)

if item not in annotated\_set:

item\_type = canvas.type(item)

fills = canvas.itemconfigure(item, *'fill'*)

if fills[-1] == *''*:

pass

elif item\_type == *"text"*:

pass

elif item\_type == *"polygon"*:

pass

else:

items\_over.append(item)

return items\_over