Introducing Programming to the Visually Impaired

One Pixel at a Time

# Introduction

I plan to provide you with a demonstration of my work which aids the visually impaired in learning programming. We add both tactile and auditory feedback to graphical python turtle programs. These are simple programs written or modified by the student. I will follow the demonstration with a discussion of the implementation. I will touch on other paths I might have taken here.

Our software enhances the user’s “viewing” experience by adding **extended graphics view**s to programs containing turtle statements. An **extended graphics view** provides an additional user sense to the standard turtle display.

We have added the following **extended graphics view** displays to each program’s normal display:

* **Text output**: rectangular array of letters to be sent to Braille machine which produces a coarse resolution picture
  + Currently 40 columns by 25 rows, representing the turtle window display
  + 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 output. 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 dots arranged to simulate actual Braille.

# A Demonstration (Hopefully Live)

Hopefully of “Hello World!” simplicity, with a bit of graphics flair. In truth, we went a bit further to better demonstrate the capabilities or limitations of our displays. We normally begin with a single-colored square.

# spokes.py

# Display a star with spokes

from turtle\_braille\_link import \* # Set link to library

#from turtle import \* # Bring in turtle graphic functions

speed(*"fastest"*)

for i in range(7): # Do things 7 times

if i == 0:

color(*"red"*)

elif i == 1:

color(*"orange"*)

elif i == 2:

color(*"yellow"*)

elif i == 3:

color(*"green"*)

elif i == 4:

color(*"blue"*)

elif i == 5:

color(*"indigo"*)

else:

color(*"violet"*)

forward(300)

dot(100)

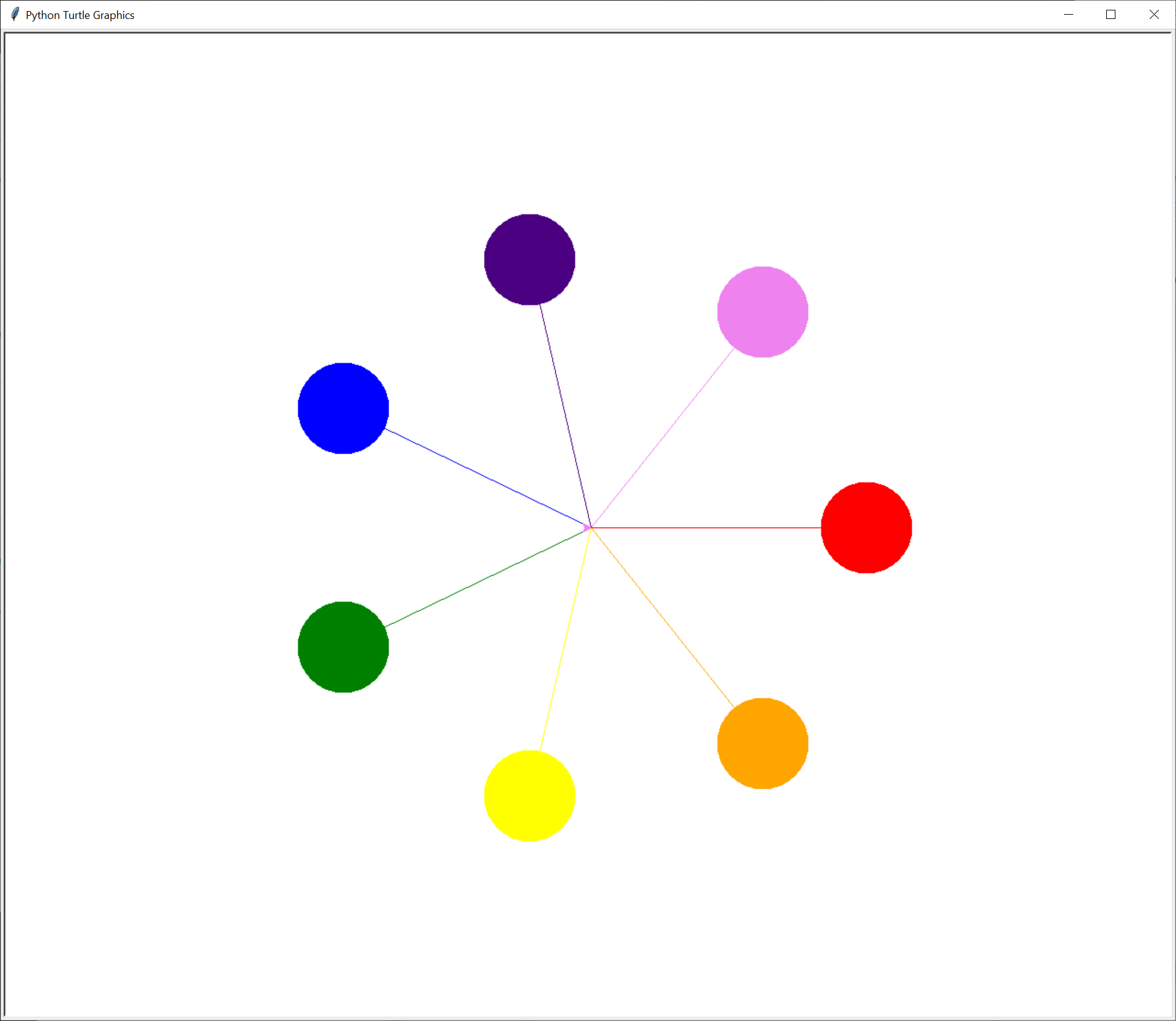
backward(300)

right(360/7)

done()

## Sample Program Output

### Turtle Graphics Screen



### Standard Output, Including Text for Brallier

resource\_lib\_proj/src is already in path

resource\_lib\_proj/src is already in path

*… Omitted for brevity …*

Braille Display - Braille Print Output

,,,,,,,,,,,,,iii

,,,,,,,,,,,,iiiiii

,,,,,,,,,,,,iiiiii,,,,,,,,vvv

,,,,,,,,,,,,iiiii,,,,,,,vvvvvv

,,,,,,,,,,,,,,,i,,,,,,,,vvvvvv

,,,bbb,,,,,,,,,ii,,,,,,,,vvvvv

,,bbbbb,,,,,,,,,i,,,,,,vvv

,,bbbbbb,,,,,,,,i,,,,,vv

,,bbbbbbbb,,,,,,ii,,,vv

,,,,,,,,,,bbbb,,,i,,v,,,,,,,,,,,rr

,,,,,,,,,,,,,bbbbivv,,,,,,,,,,rrrrrr

,,,,,,,,,,,,,,,,bvvrrrrrrrrrrrrrrrrr

,,,,,,,,,,,,,ggg,yoo,,,,,,,,,,rrrrrr

,,,,g,,,,ggggg,,,y,,o,,,,,,,,,,,rr

,,gggggggg,,,,,,yy,,,oo

,,gggggg,,,,,,,,y,,,,,oo

,,ggggg,,,,,,,,,y,,,,,,ooo

,,,,g,,,,,,,,,,yy,,,,,,,,ooooo

,,,,,,,,,,,,,,,y,,,,,,,,oooooo

,,,,,,,,,,,,yyyyyy,,,,,,oooooo

,,,,,,,,,,,,yyyyyy,,,,,,,,ooo

,,,,,,,,,,,,yyyyyy

,,,,,,,,,,,,,yyy

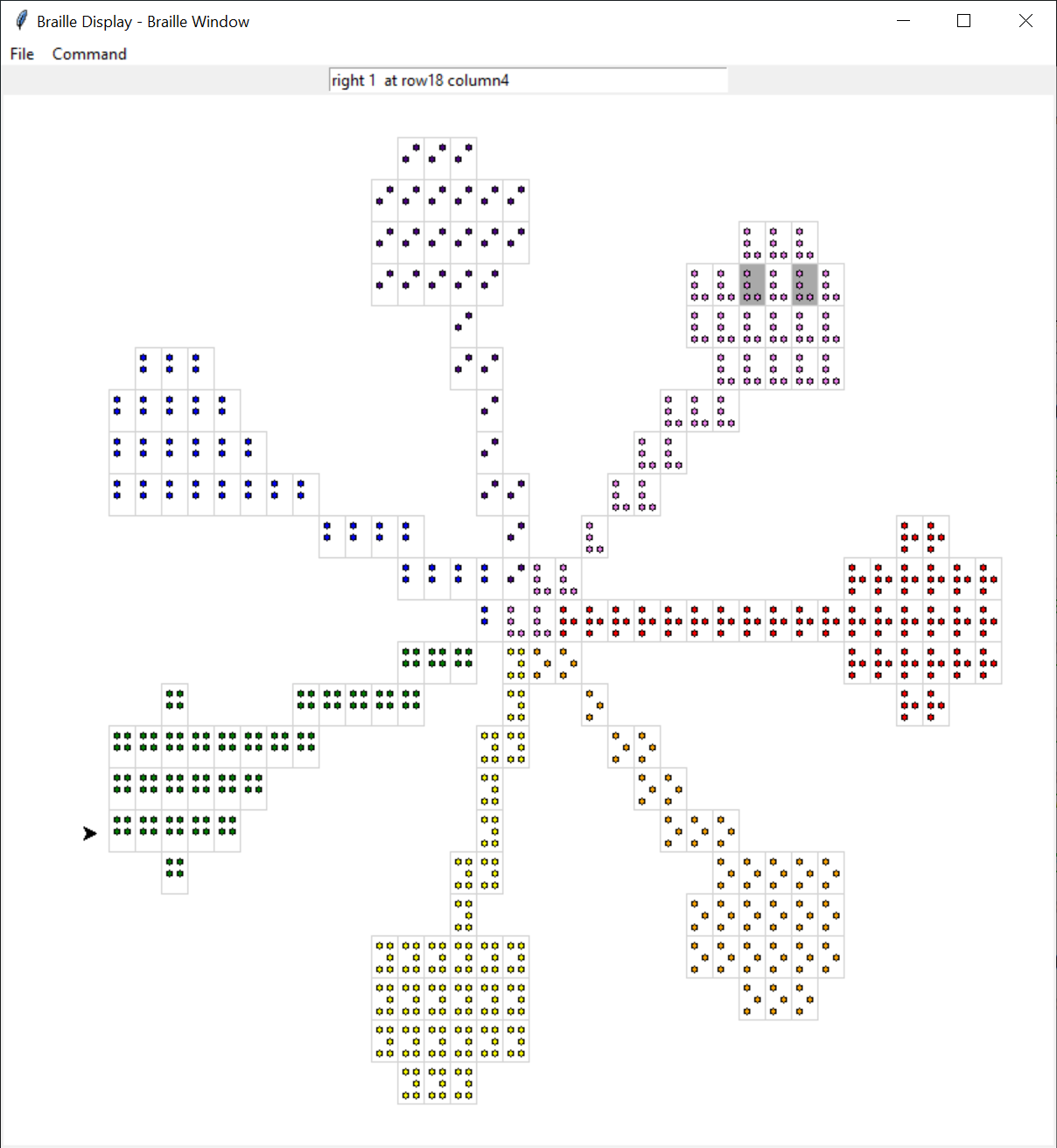
*… Omitted for brevity …*

up 2 at row17 column36

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 Braille 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 and report the content of the current cursor position. The user can, via keyboard key presses and mouse operations, move to around the window. Cursor position is reported as current color or the distance from the closest figure. Optionally reported is 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*

*g - Go to closest figure*

*p - Report/Say current position*

*Escape - flush pending report output*

Keyboard commands are case insensitive

* Up, Down, Left, Right are keyboard arrow keys
* p – Report current position (default) adds current row, column to end of position reporting messages in the form “at row *N* column *M*”
* Escape – flush pending audio output
* 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

To help with the growing number of commands, these commands were split between action-like commands such as “Up” and “Down: and state commands such as “z - stop reporting position”. The “state” commands were placed in the “Commands” menu.” These can be reached via “Alt-c”, then the corresponding short-hand letter.

Command Menu Commands

Help - list command (Alt-c) commands

h - say this help message

a - Start reporting position

z - stop reporting position

e - echo input on

o - echo off

v - visible cells

i - invisible cells

r – redraw figure

s - silent speech

t - talking speech

l - log speech

n - no log speech

p – report position

u – audio beep

d – no audio beep

Escape – flush pending report output

* a – Start reporting location (default) adds current row, column to end of position reporting messages in the form “at row *N* column *M*”
* l – log speech (default) – begin sending talk speech to standard output (console)
* n – 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 – silent - stops the talking. This may be helpful for sighted users as it greatly increases screen navigation speed.
* t – talking speech (default) starts talking reports
* v – Make figure cells visible(default) reverses w
* i – Make figure cells invisible – gives the sighted viewer a sense of the difficulty navigating when one cannot see the drawing.
* z – Stop reporting location – removes “at row *N* column *M*” from end of position reports.
* u – audio beep – Position is reported via audio beeps to reduce response delay. Cells are announced with a color determined beep frequency. Blanks have a lower frequency beep. After a cell is announced, the probable next position/cell is announced in a shorter beep to aid the user in sensing a change.
* d – no audio beep – restores talking position announcing

#### Accelerated Feedback – “audio beep”

Talking figure feedback greatly slows down display navigation. This slowdown is somewhat reduced by omitting some of the feedback such as the input echo and position announcing. In an attempt further reduce the overhead of speech we introduced an “audio beep” mode, in which the feedback was done via audio beeps produced by the Python winsound module. Blank area produces a low frequency beep. Colored cells produce a beep whose frequency indicates the color. We currently handle only the main 7 rainbow colors. Because of the limited delay of the beeps, we added a predictive beep to indicate the cell which follows, given the current search direction. We added a “p – position command” to speak the current position to makeup for lack of exact position announcements in audio beep mode.

# 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)
  + The audio beeping, used for faster location info uses the python winsound module which is Microsoft Windows specific.
* User coding
  + Use “from turtle\_braille\_link import \*” instead of “from turtle import \*”.
* Turtle feature limitations
  + 70-80% of language – no animation support
  + Limited color support
  + circle not well supported – no time
  + shape plus some others – no time

# Background

Before outlining inner details, I will give a short outline as to how this program 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 stuff but my heart isn’t in it. I scrambled to find a different 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 like a game. 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 might be curious at least at what programming can be – not just the mad typing on a console as one might see on TV or in a movie. 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 many programs, while instructive, 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, that would provide additional visual impact. To facilitate this, we made use of 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 graphical examples were much less impactful to the prospective programmer who can’t see the pictures. What’s left? Should we restrict programming examples to text-based output? This would be possible, but would, sadly, abandon a whole area of programming. Faced with the scarcity of widely available economical graphical display devices for those who can’t see, we were stuck. However, rather than trying to tackle the very large problem of general graphical presentation for the visually impaired, I decided, to concentrate on making simple graphical programs more visible.

# How It Works

## Overview

### Functional flow

Our general approach captures each turtle command, e.g., **forward**, first sending this command, unchanged, to the turtle module. Secondly the turtle command is imitated, creating a similar graphical part of an **extended graphics view**. At the end of the program display process, during the call to **done** or **mainloop**, the completed **extended graphics view** is rendered into one or more displays.

### A Detailed Flow Example – **forward** turtlecommand

Conventions used below:

* Code file names are in bold to right.

**file\_name.py**

* Timing comments are as such:

Relative execution time point: Action

* Program code snippet:

Code from file

**square.py**

0.1: Setup linkage from user written turtle statements to our program processing

from turtle\_braille\_link import \*

turtle\_braille\_link.py

0.2: Add TurtleBraille/TurtleDisplay code to path

if not in\_path:

sys.path.append(dir\_check)

0.3: Add links to shadow turtle functions

from turtle\_braille import \*

0.4: Create an instance of TurtleBraille and a link from external shadow turtle functions to the shadow turtle object.

tum = TurtleBraille()

**turtle\_braile.py**

0.5: Create an instance of BrailleDisplay, the kernel of the shadow turtle production, and the link to this kernel.

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

**square.py**

forward(200)

1: Turtle call in user program

A brief review of turtle:

* forward() – move pen forward specified amount, drawing a line.
* right() – turn right specified number of degrees.
* done() – complete drawing, continue display till done.

**turtle\_braile.py**

2: Redirect turtle to shadow turtle function of the same name.

def **forward**(length):

return tum.forward(length)

3: Direct call to BrailleDisplay member function.

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

return *self*.bd.forward(length)

braille\_display.py

4: Call turtle function **forward** to do “turtle action”, followed by doing extended graphical view work**.**

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**: Implement shadow turtle **goto**, used above to complete **forward** function.

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: Create line in **extended graphics view**, used where ever good lines are needed.

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

turtle\_braille.py

7: Complete external graphics views( inside shadow turtle **done** or **mainloop**)

def **mainloop**():

return tum.mainloop()

def **done**():

return tum.done()

class **TurtleBraille**():

… omitted for brevity …

def **mainloop**(*self*):

title = *self*.title

if title is None:

title = *"Braille Display -"*

*self*.bd.display(title=title,

braille\_window=*self*.braille\_window,

points\_window=*self*.points\_window,

braille\_print=*self*.braille\_print,

print\_cells=*self*.print\_cells)

*self*.bd.mainloop()

def **done**(*self*):

return *self*.mainloop()

def **display**(*self*, braille\_window=True,

braille\_print=True,

print\_cells=False, title=None,

points\_window=False,

tk\_items=False):

*""" display grid*

**:braille\_window:** *True - make window display of*

*braille*

*default:True*

**:braille\_print:** *True - print braille*

*default: True*

**… Other options …**

*"""*

if braille\_window:

*self*.braille\_window(title=tib)

…Other stuff…

if braille\_print:

*self*.print\_braille(title=tib)

braille\_display.py

8: Create specified displays

9: Create Braille Window with audio feedback

10. Create text picture for Braille machine

def **braille\_window\_audio**(*self*, title, show\_points=False):

*""" Display current braille in a window*

*with audio feedback*

**:title:** *window title*

**:show\_points:** *Show included points instead of braille*

*dots*

*default: False - show braille dots*

*"""*

if title is not None and title.endswith(*"-"*):

title += *" Braille Window"*

aud\_win = AudioWindow(title=title,

win\_width=*self*.win\_width,

win\_height=*self*.win\_height,

grid\_width=*self*.grid\_width,

grid\_height=*self*.grid\_height,

x\_min=*self*.x\_min,

y\_min=*self*.y\_min)

aud\_win.draw\_cells(cells=*self*.cells,

show\_points=show\_points)

def **print\_braille**(*self*, title=None):

*""" Output braille*

*"""*

if title is not None:

print(title)

if *self*.shift\_to\_edge:

*self*.find\_edges()

left\_edge = *self*.left\_edge

top\_edge = *self*.top\_edge

else:

left\_edge = 0

top\_edge = *self*.grid\_height-1

for iy in reversed(range(top\_edge)):

line = *""*

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

cell\_ixy = (ix,iy)

if cell\_ixy in *self*.cells:

cell = *self*.cells[cell\_ixy]

color = cell.color\_str()

line += color[0]

else:

line += *" "*

line = line.rstrip()

if *self*.blank\_char != *" "*:

line = line.replace(*" "*, *self*.blank\_char)

###print(f"{iy:2}", end=":")

print(line)

### 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**

### 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 whether 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*

# 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.

## Not a standard Installable Package

I should learn more about this.

# 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