Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

The PolyArea Project: Part 2

Implementing the graphic foundation and UI -- in Python

October 06, 2010
URL:http://www.drdobbs.com/open-source/the-polyarea-project-part-2/227700264

PolyArea is a GUI program written in Python that lets you to draw polygons and find their area. I developed PolyArea with the help of my son Saar and presented it in The PolyArea Project: Part 1, where I described the design and architecture of PolyArea, the implementation of the algorithm itself, and the software development practices we have used. In Part 2 I describe the graphic and UI aspect of PolyArea. It's worth noting that we have made an unorthodox choice when we picked up PyGame as our foundation library for graphics and UI.

But calculating the area of simple polygons (polygons with no holes) is not as simple as it seems. In addition, we wanted to have a nice UI where you can draw polygons on the screen and see a visualization of the algorithm in action. The complete source code is available here.

Why PyGame?

pygame is a set of Python modules for developing games and other multimedia programs that's built on top of the SDL library. The reason we picked PyGame as our foundation library for graphics and UI is simply that Saar had some familiarity with it from the book Hello World! Computer Programming for Kids and Other Beginners. PyGame is lightweight and doesn't use callbacks heavily, which makes it straightforwardto understand. However, you can still shoot yourself in the foot more easily than with a more structured framework.

PolyArea doesn't have a standard UI -- no menu, no toolbar, no buttons. You have a drawing surface where you can draw polygons with the mouse. There are a couple of keyboard commands to paint rectangles inside the finished polygon and to start over. That's about it. The accumulated area of the triangles that the polygon is composed of is displayed in the corner. PyGame is good enough for this purpose because it provides drawing functions and event handling for the mouse and keyboard.

The Main Loop

With PyGame you need to write your own main loop and spin the wheels. I wrote a generic main loop class (defined in mainloop.py) that maintains a list of display objects, handles events, handles collisions, updates the state of all the objects and renders the objects on the screen. This is pretty much what a game does. The generic main loop doesn't actually implement all these operations that some of them are specific to every game.

The class can be used as is for simple programs that just draw on the screen and don't care about events. You can just instantiate it, add objects, and call the **run()** method, which will continue to **update()** and **render()** your objects in each iteration.

The class can also be used as a base class for more complicated programs. You should subclass it with your your own class and implement the **handle_events()** and **handle_collisions()** methods.

The **__init__()** method lets you set the size of the window, caption, and background color. It defines an empty list of objects (the objects you put in this list will be rendered to the screen later). It initializes the PyGame library itself, creates a screen object (corresponds to the application window) with the requested size, fills it with the background object and sets the caption.

```
<code>
class MainLoop(object):
  def __init__(self, screen_size, caption, background_color=(255, 255, 255)):

    self.objects = []
    pygame.init()
    self.screen = pygame.display.set_mode(screen_size)
    self.screen.fill(background_color)
    pygame.display.set_caption(caption)
</code>
```

The **run()** method implements the main loop itself. It is an endless loop (while True). The program will exit though by responding to the quit event as you'll see later. The method calls in each iteration the methods for handling events (e.g., mouse clicks and key presses), updating objects, handling collisions (not used in the PolyArea program), and most importantly rendering the objects to the screen.

```
<code>
  def run(self):
    while True:
      self.handle_events(pygame.event.get())
      self._update_objects()
      self.handle_collisions()
      self._render_objects()
```

```
</code>
```

The implementation of the **handle_events()** just checks for the QUIT event and exits if it's present. As a generic main loop implementation there isn't much that can be done.

```
<code>
  def handle_events(self, events):
    """ """
    if pygame.locals.QUIT in events:
      sys.exit()
</code>
```

The **handle_collisions()** method is empty, which means you need to override it in your main loop subclass. It is not used in PolyArea because there are no moving objects. In a game, after all, the object positions have been updated you would want to handle collisions such as missiles hitting targets, characters running into walls, etc.

```
<code>
  def handle_collisions(self):
    """ """
</code>
```

The **_update_objects()** method is fully implemented. It simply iterates over all the objects and calls the **update()** method of each object. Nothing else is needed.

```
<code>
  def _update_objects(self):
    for o in self.objects:
      o.update()
</code>
```

The **_rednder_objects()** method iterates over all the objects (after they have been updated and all the collisions were handled) and calls the **render()** method of each one passing the screen object (on which the objects will render themselves). The rendering typically takes place on an off-screen buffer and then the entire off-screen buffer is displayed as one image on the actual screen by calling the **pygame.display.flip()** function.

```
<code>
  def _render_objects(self):
    for o in self.objects:
      o.render(self.screen)
    pygame.display.flip()
</code>
```

The PyGame Objects

The **BaseObject** class (defined in mainloop.py) serves as a blueprint to objects that the **MainLoop** class knows how to work with. For each type of object you want to display on the screen you should create a sub-class implement the **__init__()**, **update()**, and **render()** methods and in your program instantiate and add these objects to the **MainLoop**'s **self.objects** list. **MainLoop** will take care of the rest and call **update()** and **render()** in each iteration. The **render()** method gets a PyGame screen object and you should use PyGame functions to render the visual representation of your object on the screen.

```
<code>
class BaseObject(object):
  """A base class for visual objects managed by the main loop

  Each object has an __init__(), update() and render() methods.
  To use it sub-class BaseObject implement the methods and add
  your object to the main loop.
  """
  def __init__(self):
    """ """

  def update(self):
    """ """

  def render(self, screen):
    """ """
</code>
```

Here are the object types that PolyArea uses. They are defined in the pygame_objects.py file.

The **Line** class is a simple visual object; see Figure 1.

[Click image to view at full size]

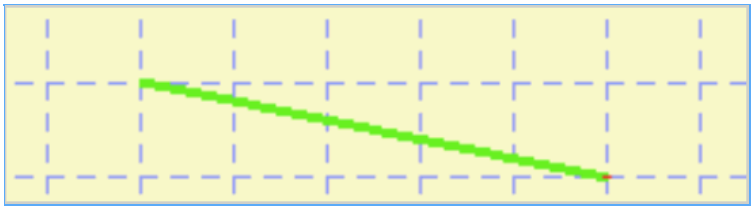

Figure 1

It subclasses **BaseObject** of course and it implements only the **__init__()** and **render()** methods. In **__init__()** it stores the color the width and the two end points of the line (**startpos** and **endpos**). In **render()** it simply calls the **pygame.draw.line()** method with the screen object and the aforementioned arguments.

```
<code>
class Line(BaseObject):
  def __init__(self, color, startpos, endpos, width=1):
    self.startpos = startpos
    self.endpos = endpos
    self.color = color
    self.width = width

  def render(self, screen):
    pygame.draw.line(screen, self.color, self.startpos, self.endpos, self.width)
</code>
```

The **Triangle** class is almost as simple as the **Line**. It draws a filled triangle with a border (Figure 2).

[Click image to view at full size]

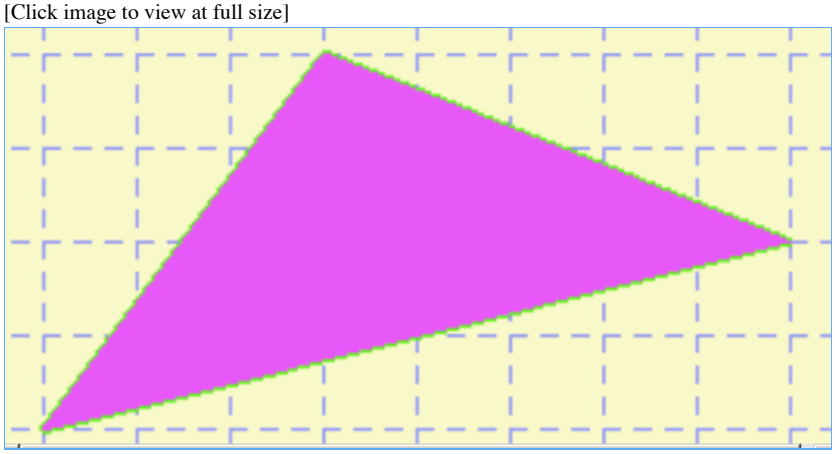

Figure 2

The **__init__()** method stores the triangle vertices (points) the border and background color and the width of the border. The **render()** method draws a filled polygon with the background color and then another non-filled polygon with the border color and width. I don't really enforce that there are three points, so you can use this class to draw any polygon, but in PolyArea I draw only triangles using this object.

```
<code>
class Triangle(BaseObject):
  def __init__(self,
               points,
               color=triangle_border_color,
               bg_color=triangle_bg_color, width=1):
    self.points = points
    self.color = color
    self.bg_color = bg_color
    self.width = width

  def render(self, screen):
    # Draw the filled interior first
    pygame.draw.polygon(screen, self.bg_color, self.points, 0)
    # Draw the border
    pygame.draw.polygon(screen, self.color, self.points, self.width)
</code>
```

People often distinguish text from graphics, but they are not that different. In the end, both are manifested as pixels on the screen. In PyGame, writing text on the screen is a two-stage process where you create a PyGame **Text** object using your favorite font, then "blit" (short "block image transfer") it to the screen at a particular position (Figure 3):

[Click image to view at full size]

Figure 3

```
</code>
class Text(BaseObject):
  def __init__(self, text, font=None, size=36, color=(0, 0, 0), pos=(0, 0)):
    self.text= text
    self.font = pygame.font.Font(font, size)
    self.color = color
    self.pos = pos

  def render(self, screen):
    text = self.font.render(self.text, 1, self.color)
    screen.blit(text, self.pos)
</code>
```

The **Grid** is pretty sophisticated compared to the other objects. It represents an interlocking grid of dashed lines with a border of three pixels. (Why three? Because I said so); see Figure 4. The **rect** parameter determines the total size of the grid. **delta_x** and **delta_y** specify the distance between vertical and horizontal grid lines and the color parameters control the color of background, border, and the dashed lines.

[Click image to view at full size]

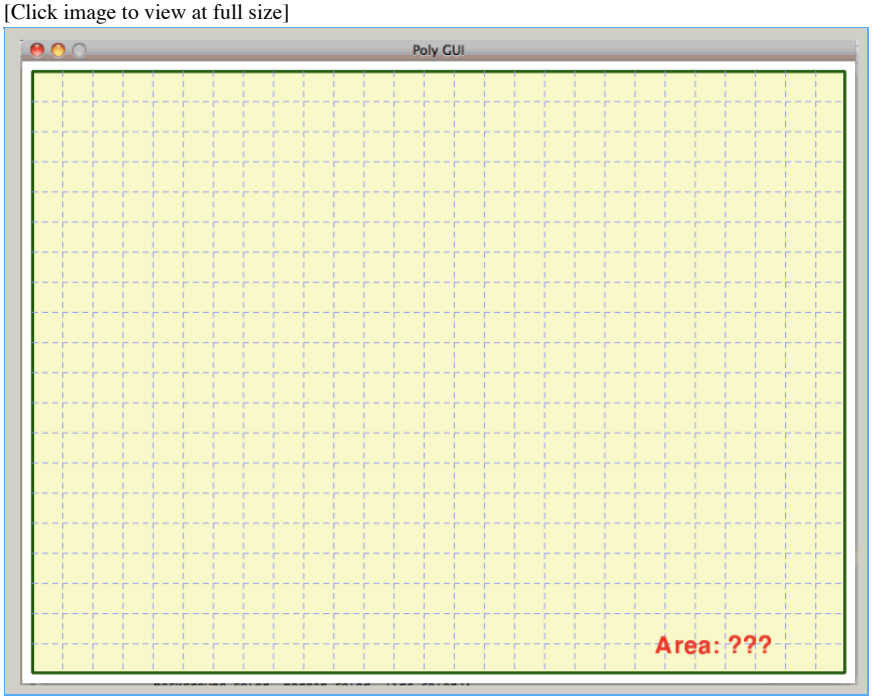

Figure 4

```
<code>
class Grid(BaseObject):
  def __init__(self, rect, delta_x, delta_y,
               background_color, border_color, line_color):
    self.rect = rect
    self.delta_x = delta_x
    self.delta_y = delta_y
    self.background_color = background_color
    self.border_color = border_color
    self.line_color = line_color

</code>
```

The **render()** method does a lot. It's not a couple of calls to **pygame.draw.*()** functions anymore. The grid is made of dashed lines, which PyGame doesn't support out of the box. So, the **render()** method defines a nested helper function called **dashed_line()** that draws a dashed line made of 5-on/5-off pixel segments. Each "on" segment is drawn using **pygame.draw.line()**. To simplify things, **dashed_line()** can draw either horizontal or vertical dashed lines (that's all I need for the grid):

```
<code>
  def render(self, screen):
    """Draws a grid on the screen"""
```

```
    def dashed_line(screen, line_color, start_pos, end_pos):
      """Draw a dashed line (must be horizontal or vertical)"""
      horiz_line_count = (end_pos[0] - start_pos[0])/10
      vert_line_count = (end_pos[1] - start_pos[1])/10
      if vert_line_count == 0:
        #draw horizontal lines
        for i in range(horiz_line_count):
          y = start_pos[1]
          x1 = i * 10 + start_pos[0]
          x2 = x1 + 5
          pygame.draw.line(screen, line_color, (x1, y), (x2, y))
      elif horiz_line_count == 0:
       #draw vertical lines
       for i in range(vert_line_count):
          x = start_pos[0]
          y1 = i * 10 + start_pos[1]
          y2 = y1 + 5
          pygame.draw.line(screen, line_color, (x, y1), (x, y2))
      else:
        raise Exception('Line must be horizontal or vertical')
</code>
```

The **render()** method follows a simple flow of erasing the contents of the grid (that facilitates dynamic changes to objects rendered on top of it), drawing the border and then the grid lines. The grid lines are drawn using the **dash_line()** function. First all the horizontal lines are drawn separated by **delta_y** pixels from each other and then all the vertical lines.

```
<code>
    # Erase the contents of the grid
    pygame.draw.rect(screen, self.background_color, self.rect)

    # Draw the grid's border
    pygame.draw.rect(screen, self.border_color, self.rect, 3)

    # Draw the grid lines
    y = self.rect.top + self.delta_y
    x = self.rect.left + self.delta_x
    while y < self.rect.bottom:
      dashed_line(screen, self.line_color, (self.rect.left, y), (self.rect.right, y))
      y += self.delta_y
    while x < self.rect.right:
      dashed_line(screen, self.line_color, (x, self.rect.top), (x, self.rect.bottom))
      x += self.delta_x
</code>
```

The Interactive UI

Okay, all the preliminaries are out of the way. You now understand how PyGame is used to draw simple geometric shapes on the screen and how visual objects can be created and displayed using the simple framework of the generic main loop and the **BaseObject** sub-class. It's time to delve into the actual interactive user interface part and see how it operates hand-in-hand with the algorithmic core discussed in Part 1.

The PolyArea UI

The user interface of PolyArea is very simple: You can draw a polygon, you can clear a polygon, and once a polygon is complete you can visualize and find its area incrementally.

Drawing a Polygon. This operation is performed with the mouse. You start by clicking anywhere on the empty grid. Now, when you move your mouse a line is drawn from the first clicked point to the current mouse position. If you click again the current line becomes permanent (part of the polygon) and from its end a new temporary line is drawn to the current mouse position. This continues until you close the polygon by clicking again on the first point. There are a few more details. The PolyArea program will constrain you to grid points only to avoid too crowded polygons that don't look nice. This also helps when you want to close the polygon. The last feature is that PolyArea will not let you create invalid polygons where lines cross (or touch) other polygon lines. You can still get yourself in trouble and find yourself in a situation where you have nowhere to go (e.g. if you create an unclosed polygon that covers all the grid points). It is pretty easy to avoid it and if worse come to worst you can always clear the current polygon and start all over.

Clearing a Polygon. You can clear the current (potentially unclosed) polygon any time. Just press **n** (for "new") on the keyboard and the grid will be cleared.

Incrementally Finding the Area. When you close a polygon PolyArea will immediately triangulate it. Now, you can observe your handiwork by repeatedly pressing the Spacebar. PolyArea will paint a triangle every time you press the Spacebar and update the text display with the accumulated area of the triangles. Once the polygon is fully painted, further key presses on the Spacebar will do nothing. You can now clear the polygon by pressing **n** and start a new polygon.

Many aspects of the look-and-feel such window size, line widths, and colors can be customized by editing the config.py file. Fill free to play with it. It's a Python file, but it's very similar to a simple .ini file with a list of key,value pairs:

```
<code>
# Main Window
main_window_bg_color = (255,) * 3
main_window_size = (830, 620)
main_window_title = 'Poly GUI'
```

```
# Grid
grid_delta_x = 30
grid_delta_y = 30
grid_padding = 10
grid_bg_color = (255, 255, 200),
grid_border_color = (10,100,10),
grid_line_color = (150, 150, 255)

# Triangle
triangle_border_color = (255, 0, 255)
triangle_bg_color = triangle_border_color

# Area text
text_color = (255, 0, 0)
text_size = 36
text_pos = (main_window_size[0] - 200, main_window_size[1] - 50)
</code>
```

The ui.py file is the entry point to the program. When you launch it the **main()** function is called. The main function creates a **PolyMainLoop** object with the appropriate window size, title and background color, creates and adds a **Grid** object and finally creates and adds a **Text** object to display the area of the polygon (initially just "???"). Once all the preparations are done it calls the main loop's run() so it can start handling events and interact with the user. Most of the constants come from the above mentioned config.py file.

```
<code>
def main(render_func=nop):
  mainloop = PolyMainLoop(main_window_size,
                          main_window_title,
                          main_window_bg_color)
  # Create grid object
  delta_x = 30
  delta_y = 30
  padding = 10
  grid = Grid(rect=pygame.rect.Rect(grid_padding,
                                    grid_padding,
                                    main_window_size[0] - 2 * grid_padding,
                                    main_window_size[1] - 2 * padding),
              delta_x=grid_delta_x,
              delta_y=grid_delta_y,
              background_color=grid_bg_color,
              border_color=grid_border_color,
              line_color=grid_line_color )
  # Add grid to the main loop's list of objects
  mainloop.objects.append(grid)
  mainloop.objects.append(Text('Area: ???',
                               size=text_size,
                               color=text_color,
                               pos=text_pos))

  # Run the main loop
  mainloop.run()

if __name__=='__main__':
  main()

</code>
```

The PolyAreaMainLoop

This class drives the dynamic interaction of PolyArea with the user. It handles mouse and keyboard events, updates the objects list and invokes the algorithmic core when a polygon is closed. The **__init__()** method initializes the generic **MainLoop** base class and defines a bunch of state variables needed to keep track of the current polygon. Remember that the **main()** function adds the fixed **Grid** and **Text** object to the object list after **PolyMainLoop** is created.

```
<code>
class PolyMainLoop(MainLoop):
  def __init__(self, screen_size, caption, background_color=(255, 255, 255)):
    MainLoop.__init__(self, screen_size, caption, background_color)
    self.prevPos = None
    self.startpos = None
    self.oldLine = None
    self.polygon_complete = False
    self.area = 0
    self.firstPoint = None
    self.triangles = []
</code>
```

PolyMainLoop gets the **run()** implementation from its base class and just overrides the **handle_events()** method. It gets a list of PyGame events and starts processing them. If the QUIT event is received it just exits the program. There is no need to shutdown cleanly or persist anything. Mouse events are handled by dedicated methods (**_onMouseDown()** and **_onMouseMove()**). Keyboard events are checked: If Spacebar is pressed the **_paint_next_triangle()** method is called and if **n** is pressed the object list is cleared except for the **grid** and **text** objects and the internal state is reset (the **_reset()** method).

```
<code>
  def handle_events(self, events):
    for event in events:
```

```
      if event.type == pygame.QUIT:
        sys.exit()
      elif event.type == pygame.MOUSEBUTTONDOWN:
        self._onMouseDown(event)
      elif event.type == pygame.MOUSEMOTION:
        self._onMouseMove(event)
      elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_SPACE:
          self._paint_next_triangle()
        elif event.key == pygame.K_n:
          # Keep just the grid and the area text
          self.objects = self.objects[:2]
          self._reset()
</code>
```

Let's follow the normal interaction of the user with program. Nothing important happens until the first click, so the first interesting event is handled by **_onMouseDown()**. This function accepts a PyGame event object that contains the current position of the mouse.

Handling mouse clicks starts by trying to bail out early. If the polygon is already complete then there is no need to respond to mouse clicks:

```
<code>
  def _onMouseDown(self, event):
    """ """
    if self.polygon_complete:
      return
</code>

If the user clicks multiple times on the same spot only the first click should be handled.


<code>
    # Ignore repeated clicks on the same spot
    if self.prevPos == event.pos:
      return

    self.prevPos = event.pos
</code>
```

Now it tried to translate the exact mouse position to a nearby grid point by calling the **fixPoint()** method. This method is the most complicated code of the UI and it may or may not find a grid point that satisfies all the constraints in the vicinity of event.pos. If it finds a suitable grid point, it returns it (Figure 5); otherwise it returns "None". If no grid point is found, the mouse click is ignored.

[Click image to view at full size]

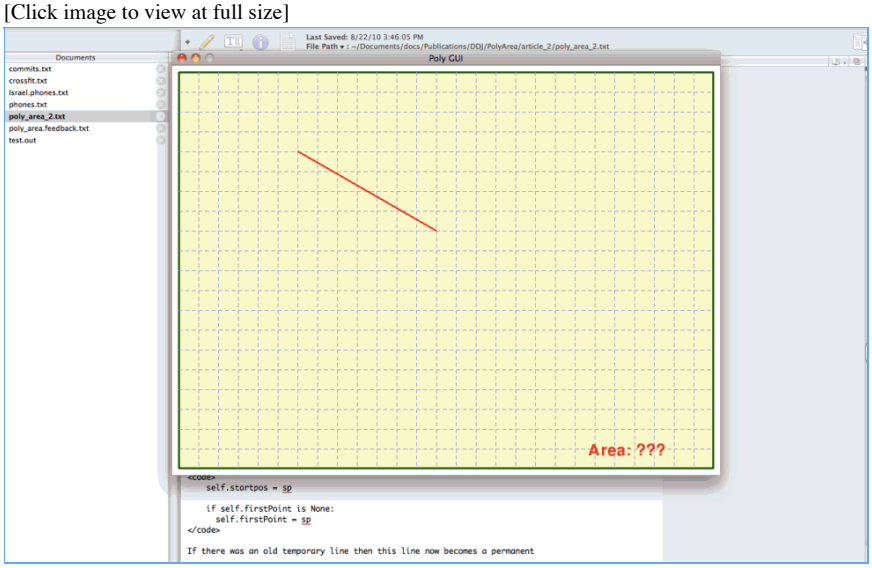

Figure 5

```
<code>
    sp = self.fixPoint(event.pos)
    if not sp:
      return
</code>
```

The last check is repeated clicks on the fixed first point. This is a special case and can happen if after the clicking for the first time the user moves the mouse just a little bit.

```
<code>
    # Ignore repeated clicks on the first point
    if self.oldLine is not None:
      if sp == self.oldLine.startpos == self.oldLine.endpos:
```

```
        return
</code>
```

At this stage, we are clear and can actually process the mouse click. **self.startpos** gets the fixed click point and becomes the start point of a new polygon line. Also, if this is the first click we store it as **self.firstPoint**.

```
<code>
    self.startpos = sp

    if self.firstPoint is None:
      self.firstPoint = sp
</code>
```

If there was an old temporary line, then this line now becomes a permanent polygon line. A **Line** object is created (going from **oldLine.startpos** to current point) and added to the object list.

```
<code>
    if self.oldLine is not None:
      line = Line(color=(0, 255, 0),
                  startpos=self.oldLine.startpos,
                  endpos=self.startpos,
                  width=3)
      self.objects.append(line)
</code>
```

If the current point is the first point, then it means the polygon has just been closed. The **oldLine** is removed (no more temporary lines) and the algorithmic core is invoked.

```
<code>
      if self.startpos == self.firstPoint:
        self.objects.remove(self.oldLine)
        self.oldLine = None
        self.polygon_complete = True
        self._do_algorithm()
</code>
```

Let's see what happens when the mouse moves.

Handling mouse motion. When the mouse moves the program can be in several states. If the current polygon is complete or if there is no polygon at all, then mouse motion can be ignored.

```
<code>
  def _onMouseMove(self, event):
    """ """
    if self.polygon_complete:
      return
    if self.startpos is None:
      return
</code>
```

If there was an old temporary line it needs to be removed.

```
<code>
    # Remove old line
    if self.oldLine is not None and self.oldLine in self.objects:
      self.objects.remove(self.oldLine)
</code>
```

If the fixed mouse position is invalid, then no temporary line will be drawn and the users will have to move their mouse to a valid position.

```
<code>
    end_pos = self.fixPoint(event.pos)
    if end_pos is None:
      return
</code>
```

At this stage everything is fine and a new **Line** object is created from the last fixed click position (**self.startpos**) to the current fixed mouse position. This **Line** object is appended to the objects list so it will be rendered on the screen. Finally, a reference to this line is stored in **self.oldLine** so it can be removed later.

```
<code>
    # Add current line to mainloop's objects
    line = Line(color=(255, 0, 0),
                startpos=self.startpos,
                endpos=end_pos,
                width=3)
    self.objects.append(line)

    # Remember the current line so it can be removed when the mouse moves
    self.oldLine = line
```

```
</code>
```

How to fix a point. When users move the mouse around or clicks it, PolyArea doesn't use the exact mouse position. Instead it looks for the nearest grid point that doesn't intersect or touch any existing polygon line. The **fixPoint()** method uses various helper methods and functions that I will not cover in detail.

The first thing it does is create a special function called **s2g** by doing partial application of the generic **snapToGrid()** function with the actual grid and the distance between columns and rows (**dx** and **dy**). This is a neat functional where you use the **functools.partial()** function to turn a function with some arguments to a function where some of the arguments are fixed.

```
<code>
  def fixPoint(self, point):
    grid = self.objects[0]
    dx = grid.delta_x
    dy = grid.delta_y

    s2g = partial(snapToGrid, dx=dx, dy=dy, grid=grid)
</code>
```

For drawing purposes the coordinate system is based on the window itself (0,0 is top left corner of the window), but for computation purposes it is better to work based on the grid coordinate system (0,0 is the top left corner of the grid) so both the target point and the start point of the current line (if there is one) are converted to the grid coordinates using the **window2grid()** function.

```
</code>
    # Normalize the point to the grid
    w2g = window2grid
    sp = w2g(self.startpos, grid) if self.startpos else None
    p = w2g(point, grid)
</code>
```

If there is no current line then simply snap the point to the grid, convert it back from grid to the window coordinate system and return it. That's the easy case.

```
<code>
    # It's the first point
    if not sp or not self.firstPoint:
      return grid2window(s2g(p), grid)
</code>
```

At this point the polygon lines are computed. There is no need to fix them because they are already on grid points, but they need to be converted to the grid coordinate system.

```
<code>
    # Find all the polygon lines
    poly_lines = [x for x in self.objects if isinstance(x, Line)]
    # Remove the oldLine
    if self.oldLine in poly_lines:
      poly_lines.remove(self.oldLine)
    # Convert polylines to grid co-ordinates
    poly_lines = [(w2g(line.startpos, grid), w2g(line.endpos, grid))
                  for line in poly_lines]
</code>
```

If the current point and its snapped to grid point don't intersect or overlap any polygon line (that's what **_checkEndPoint()** checks) then return the current point.

```
<code>
    if self._checkEndPoint(sp, p, poly_lines, s2g):
      return grid2window(s2g(p), grid)
</code>
```

If we got here it's possible the line from self.startpos to the current point intersects with other polygon lines.

```
<code>
    intersections = self._findIntersections(sp, p, poly_lines, s2g)
</code>
```

Now, if there were any intersections the code restricts the target point to the nearest intersection. The nearest intersection is found by calculating the distance to each intersection point.

```
<code>
    # If there are any real intersections
    if intersections:
      # There were intersections. Find the nearest one and restrict the point
      # to the nearest grid point that doesn't intersect.
      nearest = None
      distance = sys.maxint

      # Find the nearest intersection by iterating over
```

```
      # all the intersection points and keeping the intersection point
      # whose distance from the start position is the shortest.
      for intersection in intersections:
        d = helpers.calc_distance(intersection, sp)
        if d < distance:
          distance = d
          nearest = intersection

      xp = s2g(nearest)
</code>
```

The next step is to find all the neighboring points.

```
<code>
      #Find all potential 8 neighbors
      neighbours = [
        (xp[0] - dx, xp[1] - dy),
        (xp[0], xp[1] - dy),
        (xp[0] + dx, xp[1] - dy),
        (xp[0] - dx, xp[1]),
        (xp[0] + dx, xp[1]),
        (xp[0] - dx, xp[1] + dy),
        (xp[0], xp[1] + dy),
        (xp[0] + dx, xp[1] + dy),
        ]
</code>
```

If the corrected intersection point was on the edge or corner of the grid, then some neighbors are outside the grid and must be filtered out.

```
<code>
      # Eliminate neighbors outside the grid (if xp is on edge or corner)
      all_neighbours = neighbours[:]
      neighbours = [n for n in neighbours if
                    (0 <= n[0] < grid.rect[2] - grid.rect[0]) and
                    (0 <= n[1] < grid.rect[3] - grid.rect[1])]

      assert neighbours != []
</code>
```

Now, that the candidate list (**xp** and all its valid neighbors) has been assembled the closest valid point is selected. The distance to each candidate that passes and if its closer than the current best candidate AND if it passes the **_checkEndPoint()** method then it becomes the selected point (see Figure 6).

[Click image to view at full size]

Figure 6

```
<code>
      # Calculate the distance of each neighbour from the start point
      # and pick the closest one
      p = None
      selected_seg = None
      distance = sys.maxint

      for i, n in enumerate([xp] + neighbours):
        #d = helpers.calc_distance(n, sp)
        d = helpers.calc_distance(n, point)
        if d < distance:
          if self._checkEndPoint(sp, n, poly_lines, s2g):
            distance = d
            p = n
</code>
```

It is entirely possible that no valid neighbor will be found. In this case just return None and the current line will stay the same even though the mouse moved or the mouse click will be ignored.

```
<code>
      if not p:
        return None
</code>
```

But, if a valid point is found convert it back to the window coordinates system and return it.

```
<code>
    p = s2g(p)
    # Offset the result point back to the main window coordinate system
    p = grid2window(p, grid)

    return p
</code>
```

Gotchas

In this type of program there are many gotchas that you should be aware of and pay attention to.

Coordinate Systems. It is critical when dealing with multiple coordinate systems to make sure you are always using the correct coordinate systems. In, the case of PolyArea the window coordinate system is used for rendering, but the grid coordinate system, which is shifted by the grid padding width and height is used for computation.

The current configuration values set the grid padding to 10. That means that a point such as (40, 40) in the window coordinate system is just (30, 30) in the grid coordinate system. If you mix these coordinate systems and neglect to translate some points you get nasty bugs that are very hard to detect. The approach to address it is to explicitly identify the location in your code (e.g. the mouse event handlers) where you get input in one coordinate system and make sure you convert each point.

Fractured Pixels. The mouse events and snap to grid processing generate integral numbers. Various computations for intersections generate floating point numbers. Mixing them up can sometimes hurt you in unexpected ways. For example, the Python division operation works differently for integers and floating-point numbers.

Integer division. The integer division in Python 2.x is floor division (in Python 3 that has changed).

```
<code>
>>> 3 / 5
0
</code>
```

If you expected 0.6 you are in for a surprise.

You can find the remainder using the % (modulo) operator:

```
<code>
>>> 3 % 5
3
</code>
```

Floating-point numbers. Computers represent fractions normally as floating-point numbers. In Python, there is the **float** data type. Due to the this representation scheme (2's complement) some fractions can be represented only approximately. This can lead to unexpected results:

```
<code>
>>> 3 / 5.0
0.59999999999999998
</code>
```

Yes, you will not get 0.6 even with floating-point numbers. When mixing integers and floats, point the integers are promoted to **float**s.

Rounding. Rounding is also a potentially confusing aspect. Python has the **round()** function that rounds to the nearest integer and also the **math.floor()** and **math.ceil()**. In PolyArea I needed a special rounding operation to round an integer to the closest multiple of another integer. This was needed to find the closest grid point. I came up with this function that uses both Python's regular division operation / and the floor division // (which always does floor didivision even if some of the operands are **float**s):

```
<code>
def round_int(n, r):
  """Rounds an integer n to the closest multiple of an integer r"""
  return int(n + r / 2) // r * r
</code>
```

Dimensional reversal. Another issue to be aware of is the axis. In mathematics it is common for the Y dimension to grow up (smaller numbers are bellow larger number). It is even customary to say low and high numbers, were "high" numbers are larger. In computer graphics on the other hand the X dimension runs from left to right, but the Y dimension runs from top to bottom (a takeaway from the display hardware scanning order).

Does it matter? Well, that depends on your expectations. In the last article when I discussed the algorithm and I mentioned finding top points and removing top triangles I was using the standard math dimensions. But, when PolyArea actually renders the polygons on the screen everything is flipped on its head. That means that visually PolyArea scans from the bottom up and finds bottom points and removes bottom triangles first; see Figure 7.

[Click image to view at full size]

Figure 7

Conclusion

PolyArea was a fun project. I enjoyed working with Saar and I was surprised by how challenging it turned out to be. The program itself and the process were pretty educational and the end result is cool and entertaining.