Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

The PolyArea Project: Part 1

Calculating the area of simple polygons is not as simple as it seems

August 11, 2010
URL:http://www.drdobbs.com/windows/the-polyarea-project-part-1/226700093

This is the tale of a project I recently completed with Saar, my 13-year old son. Saar is interested in the creative aspects of computers (along with playing games, of course). He likes to play around with graphics programs like GIMP and Photoshop, create Flash animations, and compose electronic music. He has also worked on a few games using programs like YoYo Games Game Maker, created some web sites, and built some cool LEGO Mind Storm projects. As a programmer, I tried for years to nudge him towards programming with less than stellar success (my fault for pushing too hard). Recently, I managed to engage him a little more and with the help of a great book Hello World! Computer Programming for Kids and Other Beginners. We worked together through the chapters and exercises and Saar did pretty well.

Then a couple of months ago, Saar suggested that we write a program to compute the area of a polygon. I got very excited because it was much more attainable than his previous ideas (writing a 3-D galactic space strategy game). I estimated it would take us two or three weeks. However, the project turned out to be a perfect example of Hofstadter's law: "**It always takes longer than you expect, even when you take into account Hofstadter's Law.**"

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.

In Part 1 of this two-part article, I describe 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.

Overall, the project has been a great experience, involving aspects of::

- Pure "research" (looking up formulae and simple algorithms)
- Mini-algorithm implementation like finding the distance between two points, finding the area of a triangle and the intersection of two lines.
- Custom graphics programming

It brought to the front some thorney programming points that bite newbies -- things like floating-point representation, comparing integers to floating-point numbers, and division by zero when computing the slope of a vertical line. It was also a great vehicle for proper software development: how to divide a project to multiple files/modules, writing unit tests to catch bugs early in isolated environment and separation of concerns (algorithmic code vs. visualization code).

The main algorithm (decomposition of the polygon into convex polygons and each convex polygon into triangles) is interesting and demonstrates common issues with computational geometry algorithms.

The user interface is user friendly and validates the polygons that you draw on-the-fly to make sure you don't create invalid polygons (e.g. with holes or intersecting lines).

Polygons

Polygons are closed geometric shapes made of straight line segments. There are different types of polygons (concave, convex, simple, star shaped, etc). The polygons that the poly area project deals with are simple polygons -- the polygon sides don't cross themselves and contain no holes.

There are formulas for finding the area of a polygon given a set of vertices or the length of each side, but these are no fun (see http://en.wikipedia.org /wiki/Polygon). A much more educational endeavor is to find the area by dividing the polygon to smaller polygons and in particularly into triangles and then adding the area of the triangles. This is called "Polygon Triangulation" and there are several algorithms that accomplish this task. The algorithm we implement in this project is a little different for pedagogical reasons. The main motivation for this project is the creation process itself and coming up with an algorithm that is visually pleasing to track its progress.

The Poly Area Project

There are three main parts to the Poly Area Project:

- The **Algorithmic core** that deals with breaking a polygon to triangles. The algorithm uses several mini algorithms like finding the area of a triangle given its sides and the distance between two points. These mini algorithms are a perfect task for a beginner like Saar to experiment with.
- The **UI** is responsible for letting users draw the polygon and to paint it gradually when the polygon is complete. It turned out to be surprisingly complicated and it took awhile to get it right.
- **PyGame** is a cross-platform Python binding for the famous SDL (Simple DirectMedia Layer) game development library: http://www.libsdl.org/. It is used as the low-level graphics library for the project. The reason we chose PyGame and not a more standard graphics library like wxPython is that

Saar was somewhat familiar with PyGame from the "Hello World" book.

Software Development

We tried to build the program in a fairly disciplined fashion and use best practices even though it is a small project. For a newbie who only programmed several programming exercises it is a big project with multiple files and custom UI.

The programming language is, of course, Python. (Again, the complete Python source code for the project is available here.) I would go on a limb and say that Python is arguably the most scalable programming language from a learning perspective. It is one of the most readable and easy to learn programming languages, yet it can be used for enterprise-grade software projects too. It has excellent support for modular programming, object-oriented programming, good documentation, lots of third-party packages (many bindings for high-performance C/C++ libraries), great community and great IDEs. It is being used heavily by big players like Google, Nasa, YouTube (even before being bought by Google), Industrial Light and Magic. Don't forget Numenta (my company).

There are broadly speaking two schools of program design: top-down and bottom-up. Top-down means that you think about the big pieces of your program and how to connect them and then you take each piece and repeat the process of decomposition until you reach to pieces small enough to implement directly. The bottom-up approach is the opposite. You start with the smallest pieces, implement them and then combine them to bigger pieces and you keep going until you have the full-fledged program. Both approaches represent extremes that usually can't work for real-world programs (although pure bottom-up is much more practical). The approach we took in this project, which is the approach I prefer for most projects is a mostly bottom-up, but after I have in my mind the overall structure of the program. It is much easier to grow a program from small pieces that you can hook up together in flexible ways then trying to figure out all the interactions first (as required by top-down programming).

Unit testing is a staple of modern software development. Bottom-up programing lends itself naturally to unit testing because you develop small units in isolation that can be tested and then you combine them. Even-though poly area is a pretty small project we used unit testing for the algorithmic core.

The UI and PyGame support code has no tests and as a result I spent a considerable amount of time chasing bugs in the UI. Saar did a great job finding and pointing out these UI bugs. I think for a project of this size it is the right choice to test the UI manually. Automated UI testing is possible, but requires more effort and the cost/benefit ratio should be taken into account. For projects with significant UI that changes often it may be worthwhile to invest in a good automated UI test suite.

Poly Area Architecture and Organization

The architecture focuses on separating the UI code from the algorithmic code to accomodate testing and also potential replacement of the low-level graphic library (PyGame). In addition, there is a little module called config.py that lets users configure the look-and-feel.

There are three modules in the algorithmic core:

- The **polygon module** is responsible for the main algorithm. It has a custom **Polygon** class that provides some important methods used by the calc_**polygon_area()** function and its support functions: **find_Second_top_point()** and **remove_top_triangle()**. It uses functions from the triangle and helpers modules.
- The **triangle module** contains two functions: **herons_formula()** and **calc_triangle_area()**. It also has two test functions: **test_herons_formula()** and **test_calc_triangle_area()**. This is a good example of bottom up programming. It is a very small bit of functionality (calculating the area of a triangle) encapsulated in its own module with its own tests. It is very easy to implement, test and if necessary modify it (e.g. implement the calculation using a different formula).
- The **helpers module** is a support module that contains many simple and general-purpose functions that are not tied specifically to the polygon triangulation algorithm. This module can potentially be reused in other computational geometry programs. It follows the same pattern as the triangle module and has a test function for each function. Some functions are: **find_line_coeeficients()**, **calc_distance()**, and **intersect()**.

Then there is one big UI module and a config module.

- The **ui module** is responsible for all the visual aspects of poly area and the interaction with the user. It lets you draw a polygon and make sure you draw a valid polygon. Once the polygon is closed it is divided into triangles and its area is calculated and you can visualize the algorithm operation step by step by pressing the spacebar. There is a fair amount of functionality involved and the code to verify that the polygon is valid is actually more complicated then the core algorithm. The ui module uses the **mainloop** and **pygame_objects** modules.
- The **config module** resembles a Windows .ini file. It just contains some variables such as grid resolution, line thickness, and colors. The ui module reads this file and uses the values when rendering the UI.

There are two PyGame-related infrastructure modules:

- The **mainloop module** encapsulates the typical event loop of a GUI program and provides a PyGame-based implementation. In addition to some PyGame incantations it manages a list of object it renders to the screen in each iteration. These objects must support a simple interface and objects can be added/removed dynamically from the list during the program's run.
- The **pygame_objects module** contains several objects that support the mainloop interface.

Finally there is a **BaseObject** base class.

Testing

Testing is not as rigorous as I would implement in a professional project (100% coverage), but it is pretty good. I didn't implement negative tests (tests that provide bad input on purpose to make sure the code fails as expected with the proper exception and helpful error message) because writing these tests take a lot of time (usually there are many more bad inputs than good inputs) and I control the using code, so I can make sure no bad inputs are provided.

The triangle.py and helpers.py have self tests. Normally, they should be imported and used by other modules, but if you run them directly then they execute

a **test()** function. This is done via the module's **__name__** attribute.

```
if __name__=='__main__':
 test()
```

The **__name__** attribute is set by the interpreter when the module is being run directly as in:

```
python triangle.py
```

There are internal tests for the following modules:

- The tests for the **triangle module** are very simple. The **test_herons_formula()** functions makes sure that the when passing 3, 4 and 5 as the triangle sides the **herons_formula()** function correctly returns the area as 6:

  ```
  def test_herons_formula():
    a = 3
    b = 4
    c = 5
    assert herons_formula(a, b, c) == 6
  ```

- The test for the **compute_triangle_area()** module is almost the same except that it passes the vertices of an equivalent triangle:

  ```
  def test_calc_triangle_area():
    p1 = (0, 0)
    p2 = (0, 3)
    p3 = (4, 0)
    assert calc_triangle_area(p1, p2, p3) == 6
  ```

- The **helpers module** contains simple tests for all its functions. For example, here is the **test_find_line_coefficients()** function:

  ```
  def test_find_line_coefficients():
    line = ((1,0), (1,3))
    assert find_line_coefficients(*line) == None

    line = ((0,0), (5,5))
    assert find_line_coefficients(*line) == (1, 0)

    line = ((4,5), (5,5))
    assert find_line_coefficients(*line) == (0, 5)
  ```

There is a standalone test (in its own module) for the polygon module: **polygon_test**. The reason this test is separate is that the polygon module has relatively a lot of code and cluttering it with test code would make it harder to navigate and less readable. The polygon_test module has multiple test functions for different aspects of the polygon module and a central **test()** function that runs all the specific test functions:

```
def test():
  test_is_triangle()
  test_split()
  test_find_second_top_point()
  test_remove_top_triangle()
  test_calc_polygon_area()

if __name__=='__main__':
  test()
  print 'Done.'
```

To test the polygon module properly different types of polygons are used as test subjects. The same group of polygons is used for multiple tests. The test polygons are defined globally at the beginning of the module as a set of vertices:

```
triangle = ((0,0), (0,3), (4,0))
rectangle = ((0,0), (0,3), (4,3), (4,0))
parallelogram = ((0,0), (3,1), (4,5), (1,4))
concave = ((0,0), (3,6), (6,0), (3,3))
concave2 = ((0,0), (4,6), (4,1), (5,3), (5,0))
...
concave12 = (0,1), (1,0), (1,1), (2,0), (2, 2)
```

When the test polygons became more complicated I added a little sketch comment to some of them. That helped a lot visualizing what happens during the algorithm execution. For example, here is the concave11 test polygon:

```
"""
     o
   /   \
 /      \
o   o    o
 \  | \  |
  \ |  \ |
   o     o

"""
concave11 = ((0,1), (1, 2), (2, 1), (2, 0), (1, 1), (1, 0))
```

Inside each test function **Polygon** objects are instantiated using the test polygon vertices. Here is the **test_is_triangle()** function:

```
def test_is_triangle():
  """ """
  p = Polygon(triangle)
  assert p.is_triangle()

  p = Polygon(rectangle)
  assert not p.is_triangle()

  p = Polygon(parallelogram)
  assert not p.is_triangle()

  p = Polygon(concave)
  assert not p.is_triangle()

  p = Polygon(concave2)
  assert not p.is_triangle()
```

Existing Algorithms

There are several algorithms to find the area of a polygon with and without triangulation:

- **Direct area calculation**. There are formulas for directly calculating the area of a simple polygon given its vertices or given the length of its sides and the exterior angles are known. I will not repeat them here because they don't provide an interesting programming target. You can read about it here: http://en.wikipedia.org/wiki/Polygon#Area_and_centroid
- **Polygon triangulation**. Polygon triangulation is an algorithmic process. One well know algorithm is called "Ear clipping". The idea is that every simple polygon has at least one "ear", where an ear is a triangle that's wholly contained inside the polygon and whose vertices are three consecutive vertices of the polygon. Once you find such an ear you can remove it and end up with a polygon with one less vertex. If you repeat this process you will end up with a triangle (the last ear) and then you're done. This is an elegant algorithm with a non-trivial theoretical background (the proof that any simple polygon always has ears is not trivial). Another algorithm uses sweep lines to decompose the original polygon to monotone polygons (http://en.wikipedia.org/wiki/Monotone_polygon), which are easy to triangulate.
- **The poly area algorithm** is a hybrid of these two algorithms using a divide-and-conquer approach. It is looking for an ear to clip, but it's not looking very hard. If it runs into trouble it splits the polygon into two separate polygons and starts all over with each one of the smaller polygons. Over time the polygon may be split into multiple sub-polygons and the algorithm manages all of them and collectes all the removed triangles.

Terminology

I will use the following terms to describe the algorithm so let's define them clearly.

- **Top point**: The polygon vertex with the highest Y coordinate. If there are multiple vertices with the same high Y coordinate, then the vertex with lowest X coordinate (leftmost) is the top point.
- **Second top point**: The vertex with the highest Y coordinate that is smaller than the top point's Y coordinate. If there are multiple such points then one of them is selected based on some rules. Note, that there may be vertices that are not the top point whose (Y coordinate is equals to the top point's Y coordinate) that are above the second top point.
- **Top triangle**: There are two cases here. The first case is if the top point is followed by a point whose Y coordinate is smaller. That means the top point is a spike (see Figure 1).

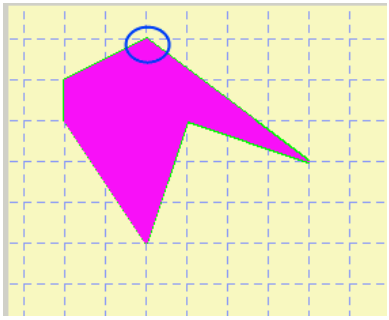

Figure 1

The second case is that the top point is followed by a point with the same Y coordinate and it forms a plateau at the top (see Figure 2). Note that the top point is always preceded by a point with a lower Y coordinate (emergent property from the definition of the top point).

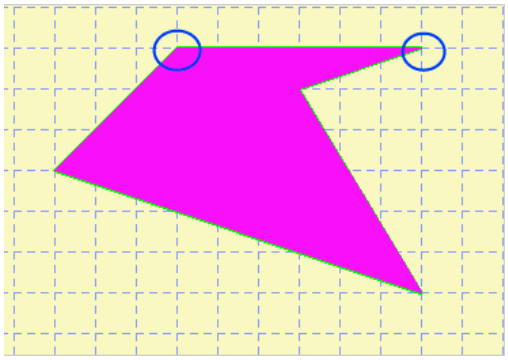

Figure 2

In the first case the top triangle will consist of the top point and the two points that are the intersection of a horizontal sweep line that goes through the second top point and the two sides of the polygon that meet at the top point. The second top point may or may not be in the top triangle (see Figure 3).

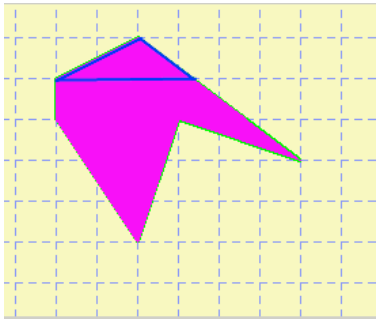

Figure 3

In the second case the top triangle consists of the top point, the point that follows it and the intersection of the left side of the polygon that ends in the top point with the sweep-line that goes through the second top point. Again, this intersection point may or may not be the second top point itself (see Figure 4)

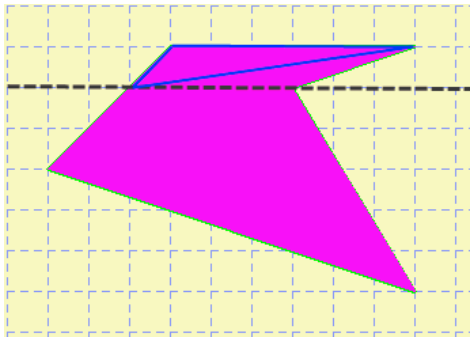

Figure 4

The Algorithm

The algorithm works by slicing triangles off the top of the polygon repeatedly until only a single triangle is left. During this process the polygon may split into two separate polygons. In this case the same process continues on both polygons (which may split too). Overall, the algorithm manages a list of polygons and in each step it slices off a triangle from one of them and potentially splits one of the polygons into two polygons.

In high-level pseudo code the algorithm can be described like this:

```
- Find top point
- Find second top point
- Slice off the top triangle
- If the second top point resides between the two lower vertices of the top
  triangle (happens only when there is no top plateau) split the polygon into
  two polygons along the segment from the top point to the second top point (see Figure 5)
- Repeat for each polygon until it is reduced to a single triangle.
```

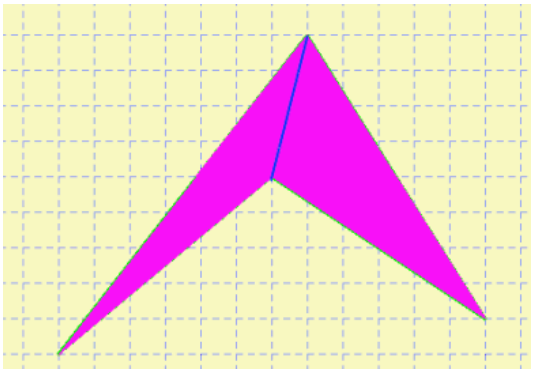

Figure 5

The Code

The algorithm is implemented in the **calc_polygon_area()** function of the polygon.py module. The function takes a list of polygons and a callback function and it starts to decimate the polygons according to the logic described above. Initially, the list of polygons will contain just the original polygon. As splits occur the list may grow up. As sub-polygons are exhausted the list of polygons may shrink until there are no more polygons. If a polygon is reduced to a single triangle it is removed from the list.

The callback interface is intended for interactive programs where the main program may do something each time a new triangle is removed from a polygon. The callback function receives the removed triangle (three vertices) and its area.

Here is the code:

```
def calc_polygon_area(polygons, callback):
  while polygons:
    poly = polygons[0]
    poly.invariant()
    if poly.is_triangle():
      polygons = polygons[1:]
      triangle = poly.points
    else:
      second_top_point, kind = poly.find_second_top_point()
      if kind == 'inside':
        # need to split the polygon along the diagonal from top to second top
        new_polygons = split(poly, (poly.sorted_points[0], second_top_point))
        polygons = new_polygons + polygons[1:]

        # No callback in this iteration because the first polygon was split
        # but no triangle was removed.
        continue

      # If got here then the target polygon (poly) has a top triangle that can be
      # removed
      triangle = remove_top_triangle(poly, second_top_point, kind)
    # Call the callback with the current triangle
    triangle = round_points(triangle)
    callback(triangle, calc_triangle_area(*triangle))

  # No more polygons to process. Call the callback with None, None
  callback(None, None)
```

The **calc_polygon_area()** function operates on **Polygon** objects. These objects are instances of the custom **Polygon** class that has some special features for the purpose of this very algorithm. For example, it manages the polygon points in a sorted list from top to bottom and points with the same Y coordinate are ordered from left to right. This helps finding the top point (it's simply the first point) and the second top point. Let's go over the **Polygon** class.

The **__init__()** method accepts a list of points (the polygon vertices), stores them (after rounding) and then sorts them and verifies that everything is valid (the **invariant()** method). The **round_points()** function was imported from the helpers module and just rounds each co-ordinate to three decimal digits, so it is more readable. It's not really necessary for the operation of the algorithm.

```
class Polygon(object):
  def __init__(self, points):
    self.points = round_points(points)
    self.sort_points()
    self.invariant()
```

The **sort_points()** method sorts the original points according to the "topness" order as needed by the main algorithm. Points with higher Y coordinate come before points with lower Y coordinate and within the points with the same Y coordinate, points with lower X coordinate come first. Python objects can be sorted using Python's **sorted()** function that take a sequence and sorts it. By default, the sequence elements are compared directly, but you can provide a custom compare function (or a function that creates a comparison key from each element). The **sort_points()** method defines an internal function called **compare_points()** that implements the "topness" order, uses it as the **cmp** argument to the built-in **sorted()** function and assigns the result to **self.sorted_points**.

```
  def sort_points(self):
```

```
  def compare_points(p1, p2):
    if p1 == p2:
      return 0
    if p1[1] > p2[1]:
      return -1
    if p1[1] < p2[1]:
      return 1
    # Same y-coordinate
    assert p1[1] == p2[1]
    if p1[0] < p2[0]:
      return -1
    else:
      assert p1[0] != p2[0]
      return 1
  self.sorted_points = sorted(self.points, cmp=compare_points)
```

The **invariant**() method verifies that there are more than two points, that there are no duplicate points, that the sorted points are sorted properly, that no three consecutive points are on the same line (has the same X or Y coordinates) and finally that there are no vertices that reside on a polygon side:

```
def invariant(self):
  assert len(self.points) > 2
  for p in self.points:
    assert len(p) == 2

  for i, p in enumerate(self.sorted_points[1:]):
    p2 = self.sorted_points[i]
    assert p2[1] >= p[1]
    if p[1] == p2[1]:
      assert p2[0] < p[0]

  # Make sure there are no duplicates
  assert len(self.points) == len(set(self.points))

  # Make sure there no 3 consecutive points with the same X or Y coordinate
  point_count = len(self.points)
  for i in xrange(point_count):
    p = self.points[i]
    p1  = self.points[(i + 1) % point_count]
    p2  = self.points[(i + 2) % point_count]
    assert not (p[0] == p1[0] == p2[0])
    assert not (p[1] == p1[1] == p2[1])

  # Make sure no vertex resides on a side
  sides = []
  for i in xrange(len(self.points)):
    sides.append((self.points[i], self.points[(i+1) % len(self.points)]))

  for p in self.points:
    for side in sides:
      if p != side[0] and p!= side[1]:
        assert not helpers.point_on_segment(p, side)
```

The most interesting and complicated method of the Polygon class is **find_second_top_point**(). It returns a pair (2-tuple) that consists of the second top point itself and its kind. There are three kinds of second top points: 'vertex', 'inside' and 'outside'. I will explain the code bit by bit because there is a lot to take in. The first stage is preparation only. The top point and the candidates for second top point are found by iterating over the **self.sorted_points** list.

```
def find_second_top_point(self):
  top = self.sorted_points[0]
  top_y = top[1]
  second_top_y = None
  second_top = None
  candidates = []
  # Find the Y co-ordinate of the second top point and all the candidates
  for p in self.sorted_points[1:]:
    if p[1] < top_y:
      if second_top_y is None:
        second_top_y = p[1]
      else:
        if p[1] < second_top_y:
          break # finished with second top candidates
      candidates.append(p)
```

The next stage is finding the vertices that are adjacent to the top point. This is needed because if the second top point is one of them then it means its kind is 'vertex'.

```
  index = self.points.index(top)
  pred = self.points[index-1]
  post = self.points[(index+1) % len(self.points)]
  assert None not in (pred, post)
```

Once you have the candidates for second top point and the **pred** and **post** points. You can start looking for 'inside' second top point. If there is a candidate second top point horizontally between pred and post then return it. There are three cases: both **pred** and **post** are candidates, only **pred** is a candidate or only **post** is a candidate. Note, that technically this point may be a vertex, but it is still classified as 'inside' in order to split the polygon (otherwise the situation gets complicated).

```
# If both pred and post are candidates and there is another candidate
# between them then pick the candidate in between as an 'inside' point
if pred in candidates and post in candidates:
  pred_index = self.sorted_points.index(pred)
  post_index = self.sorted_points.index(post)
  if abs(post_index - pred_index) > 1:
    # there is a candidate between pred and post
    index = min(pred_index, post_index) + 1
    assert index < max(pred_index, post_index)
    p = self.sorted_points[index]
    assert p in candidates
    return (p, 'inside')

# If either pred or post are candidates and there is another candidate
# between them then pick the candidate in between as an 'inside' point
if pred in candidates:
  # Find the point p on (top, post) where y = second_top. If there is a
  # candidate whose X coordinate is between pred.x and p.x then it is the
  # second top point and it's an 'inside' point'
  p = helpers.intersect_sweepline((top, post), second_top_y)
  if p is not None:
    left_x = min(pred[0], p[0])
    right_x = max(pred[0], p[0])
    for c in candidates:
      if left_x < c[0] < right_x:
        return (c, 'inside')

if post in candidates:
  # Find the point p on (top, pred) where y = second_top. If there is a
  # candidate whose X coordinate is between post.x and p.x then it is the
  # second top point and it's an 'inside' point'
  p = helpers.intersect_sweepline((top, pred), second_top_y)
  if p is not None:
    left_x = min(post[0], p[0])
    right_x = max(post[0], p[0])
    for c in candidates:
      if left_x < c[0] < < right_x:
        return (c, 'inside')
```

At this point, the option of an 'inside' point when **pred** or **post** are candidates has been ruled out and the second top point is the first candidate. If it is also either **pred** or **post** then it is a 'vertex'.

```
second_top = candidates[0]
assert second_top[1] < top_y

# If the second top point is either pred or post then it is a 'vertex'
if second_top in (pred, post):
  return (second_top, 'vertex')
```

If **pred** or **post** are at the same height as the top point (plateau, remember?) but the second top point is not pred or post (otherwise you wouldn't get here because it will return at one of the previous checks) then it is outside.If it is higher then lower between **pred** or **post** (the one that is not in the plateau). Otherwise, it is inside.

```
# If pred or post are at the same height as top then second top is 'outside'
# if the second top is vertically between pred and post, otherwise it's
# inside
if max(pred[1], post[1]) == top_y:
  if min(pred[0], post[0]) < second_top[0] < max(pred[0], post[0]):
    return (second_top, 'inside')
  else:
    return (second_top, 'outside')
```

At this stage, there is no plateau. Both **pred** and **post** are below the top point. The second top point is not **pred** or **post**. If the second top point has the same Y coordinate as pred or post then it must be outside. How come? If it was between **pred** and **post** then it can't be the leftmost point between the three.

```
# Check if pred or post are at the same height as the second top point.
# If this is the case then the second top point must be outside.
if second_top[1] in (pred[1], post[1]):
  return (second_top, 'outside')
```

Now, you know that the second top point has a higher Y coordinate, than both **pred** and **post**, but you are not sure if it is inside or outside (vertex was ruled out earlier). To figure it out you have to check the intersection of the horizontal sweep-line that with **Y=second_top** with the line segments (top, pred) and (top, post). To do that **pred** and **post** are replaced with the corresponding intersection points:

```
pred = helpers.intersect_sweepline((pred, top), second_top[1])
assert pred is not None
post = helpers.intersect_sweepline((top, post), second_top[1])
assert post is not None
```

The idea is that now you have again three points: **pred**, **post**, and **second_top** that all have the same Y coordinate and you can determine by their X coordinate if **second_top** is between **pred** and **post**(inside) or outside:

```
# Pred and post are both fixed to be on the sweepline at this point
# if they weren't already. Find their left and right X-coordinate
```

```
    left_x = min(pred[0], post[0])
    right_x = max(pred[0], post[0])

    # If the second_top_point is between post and pred it's internal
    # otherwise it's outside
    if left_x < second_top[0] < right_x:
      kind = 'inside'
    else:
      kind = 'outside'
    return (second_top, kind)
```

Okay, so you found the second top point and classified it as 'vertex', 'inside' or 'outside'. If it's 'inside' then a triangle can't be removed at the moment and you need to split the polygon into two separate polygons along the diagonal (top, second_top), which is guaranteed not to cross any other polygon line. This is exactly the job of the **split()** function. Here is how it works:

Let's say the polygon has 8 vertices numbered 0 through 7 and the diagonal runs from 3 to 6. Then the vertices from 3 to 6 (3, 4, 5, 6) will be one polygon and the vertices from 6 to 3 (6, 7, 0, 1, 2, 3) will be the second polygon. These two new polygons are kind of Siamese twins connected along the diagonal and share the diagonal vertices, but no other vertex is shared. Each twin is again a simple polygon and they don't overlap:

```
def split(poly, diagonal):
  """Split a polygon into two polygons along a diagonal

  poly: the target simple polygon
  diagonal: a line segment that connects two vertices of the polygon

  The polygon will be split along the diagonal. The diagonal vertices will
  be part of both new polygons

  Return the two new polygons.
  """
  assert type(diagonal) in (list, tuple)
  assert len(diagonal) == 2
  assert diagonal[0] in poly.points
  assert diagonal[1] in poly.points
  assert diagonal[0] != diagonal[1]

  index = poly.points.index(diagonal[0])
  poly1 = [diagonal[0]]
  for i in range(index + 1, len(poly.points) + index):
    p = poly.points[i % len(poly.points)]
    poly1.append(p)
    if p == diagonal[1]:
      break;

  index = poly.points.index(diagonal[1])
  poly2 = [diagonal[1]]
  for i in range(index + 1, len(poly.points) + index):
    p = poly.points[i % len(poly.points)]
    poly2.append(p)
    if p == diagonal[0]:
      break;

  return [Polygon(poly1), Polygon(poly2)]
```

If the second top point is not 'inside' then you can remove a triangle from the current polygon. This is the job of the **remove_top_triangle()** function. It has to handle two cases: the single top point and the plateau case. The function's doc comment in the code explains everything quite clearly, but I'll describe it here in a little more detail. The input to the function is the target polygon, the second top point and its kind. The first few lines just verify the input:

```
def remove_top_triangle(poly, second_top_point, kind):
  """  """
  poly.invariant()
  assert not poly.is_triangle(), 'Polygon cannot be a triangle'
  assert second_top_point in poly.points
  assert kind in ('vertex', 'outside'), 'Inside second top point is not allowed'
```

Next, it gets the top point and its index and verifies the top point is indeed above the second top point:

```
# Get the top point and its index
  top_point = poly.sorted_points[0]
  index = poly.points.index(top_point)

  # Make sure the top point is really above the second top point
  assert top_point[1] > second_top_point[1]
```

Then, the sweep-line is created. This is a horizontal line whose Y co-ordinate is the same as the second top point:

```
  # Create the sweepline
  x1 = -sys.maxint
  x2 = sys.maxint
  second_top = second_top_point[1]
  sweepline = ((x1, second_top), (x2, second_top))
```

Once, all the preliminary checks are done and the sweep-line is defined it's time to check if the polygon has a top plateau or a single top point. The first step is to find the vertices that precede and follow (index-wise) the top point. Care must be taken not to run out of bounds.

```
  next_point = poly.points[(index + 1) % len(poly.points)]
  prev_point = poly.points[index - 1]
```

It's time to check if we are in dealing with case 1 (two consecutive top points) or 2 and handle it. An empty list called **new_points** will store the new vertices that may need to be added to the polygon and later redundant vertices will be removed.

```
  # check if we are in case 1 (two consecutive top points) or 2
  new_points = []
```

In case 1 there is a single new point, which is the intersection of the segment from the non-plateau point with the sweep-line (if the other point is the second top point it stays as is). The second plateau point (not the top point) is added too.

```
  if max(next_point[1], prev_point[1]) == top_point[1]:
    # Case 1 - platuae
    p = next_point if next_point[1] < prev_point[1] else prev_point
    other_point = prev_point if next_point[1] < prev_point[1] else next_point
    segment = (p, top_point)
    new_point = helpers.intersect(segment, sweepline)
    assert new_point is not None
    new_points = [new_point, other_point]
```

In case 2 the new points are the intersection of the prev and next segments with the sweep-line (if one of the points is a second top point it stays as is).

```
else:
    # Case 2 - single top point

    for p in prev_point, next_point:
      if p[1] == second_top_point[1]:
        new_point = p
      else:
        segment = (p, top_point)
        new_point = helpers.intersect(segment, sweepline)
        assert new_point is not None

      new_points.append(new_point)
```

At this stage, the top triangle to be removed is constructed from the top point and the two new points.

```
  assert len(new_points) == 2
  triangle = new_points + [top_point]
```

The new points are rounded and the ones that are not in the current polygon are added to the polygon (injected where the top point used to be).

```
  new_points = round_points(new_points)
  to_add = [p for p in new_points if p not in poly.points]

  poly.points = poly.points[:index] + to_add + poly.points[index+1:]
```

The last and most delicate part of the function is to remove invalid points that violate the polygon invariant. In order to figure out what points are invalid you need to create a list of the polygon sides:

```
  # Find the polygon sides
  sides = []
  for i in range(0, len(poly.points) - 1):
    sides.append((poly.points[i], poly.points[i+1]))
  sides.append((poly.points[-1], poly.points[0]))
```

The first type of invalid point is a left vertex of the top triangle if the top triangle has a horizontal bottom and it is sticking to the left (see Figure 6). This previously valid point became invalid as a result of removing the top triangle.

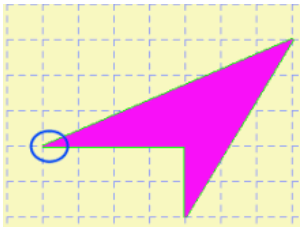

Figure 6

```
# Remove left vertex of triangle if it has a horizontal bottom and it is
# on a polygon side
if new_points[0][1] == new_points[1][1]:
  left_vertex = (min(new_points[0][0], new_points[1][0]), new_points[0][1])
  right_vertex = (max(new_points[0][0], new_points[1][0]), new_points[0][1])

  for side in sides:
    if helpers.point_on_segment(right_vertex, side):
      if right_vertex not in side:
        poly.points.remove(left_vertex)
```

Another case involes three consecutive points along the same horizontal or vertical line.

```
poly.points = helpers.filter_consecutive_points(poly.points)
```

The last case are points that end up in the middle of an existing polygon side. You need to find the polygon sides again because the polygon has been potentially modified.

```
# If there are points that are in the middle of a side remove them
to_remove = []
# Find the polygon sides again
sides = []
for i in range(0, len(poly.points) - 1):
  sides.append((poly.points[i], poly.points[i+1]))
sides.append((poly.points[-1], poly.points[0]))

# Iterate over all the polygon points and find the points
# that intersect with polygon sides (but not end points of course)
for i in range(0, len(poly.points)):
  p = poly.points[i]
  for side in sides:
  # If p is on segment s then should be removed
    if p != side[0] and p != side[1]:
      if helpers.point_on_segment(p, side):
        to_remove.append(p)

for p in to_remove:
  poly.points.remove(p)
```

Finally, sort the polygon points, verify that the new polygon still maintains the invariant and return the removed triangle.

```
poly.sort_points()
poly.invariant()
return triangle
```

Conclusion

What started off as a "little" project turned up to be more complicated than expected. I hoped for a little elegant algorithm, but it ended up as a fairly complex beast fractured into multiple cases that need to be handled separately. It was a lot of fun working together with Saar and I think that now he at least understand the complexity of software and how much work it takes to make seemingly simple things work reliably. In the next installment I will describe the poly area user interface that also turned out out to be surprisingly difficult to get right.