An stroking procedure to accomodate different alignments

Juan Luis Boya García

RFC ntrrgc@gmail.com

Here is described a procedure to stroke shapes with two new alignments: inset and outset.

Note: This procedure does not cover left and right alignments. Actually those do not require most of the steps detailed here. For right alignment is enough to calculate the outline of the curve (e.g. with `half_outline()`) and stroke it. Left alignment is performed in the same way, reversing the path first.

I. Procedure

I. Step one: Division

On a first step, perform division of the shape with a rectangle bigger than the shape (ideally infinite, but shape size plus stroke width plus miter limit would do the trick). As the result you get many non self intersecting shapes, that we will call here *subshapes*.

Figure 1 shows an example shape with thin stroke and fig. 2 shows the result of applying rectangle division to that shape. We obtain five subshapes, shown each in a different color.

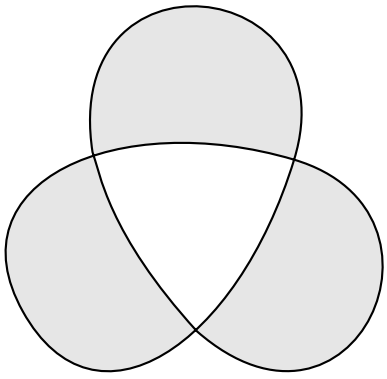

Figure 1: *A relatively simple self-intersecting shape.*

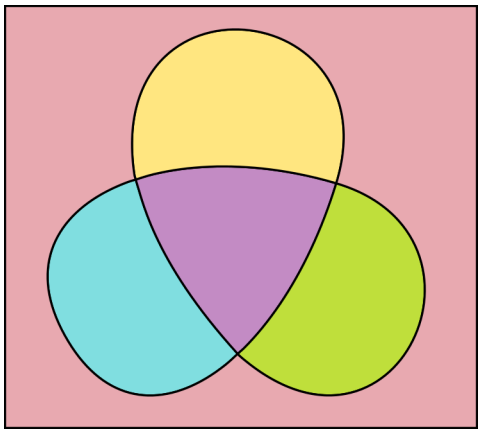

Figure 2: *The subshapes resulting from applying division.*

The outermost subshape corresponding to the rectangle is no longer needed. The rest of the procedure must not take it into account.

The subshapes themselves are not enough information though. Each point of self-intersection of the original shape must be calculated and linked to one vertex of each of the subshapes that touch it. For the example shape, those points are shown in yellow in fig. 3. The outermost subshape from the rectangle has been removed too from that figure.

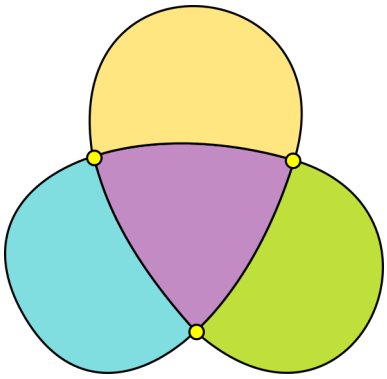

Figure 3: *Subshapes share the intersection points of the original shape.*

We will refer to each of those vertices as *intersection vertices*. Additionally, we will refer to the curve path between a pair of intersection vertices of the same subshape as *boundaries*.

In the example shape we have four subshapes: one for the purple curved triangle at the center and one for each of the three "leaves". The curved triangle at the center has three intersection vertices and three boundaries. Each of the leaves have two intersection vertices and two boundaries.

A boundary can lie on top (or *coincide with*) of another boundary of another subshape. For example, the bottom boundary of the upper leaf coincides with the upper boundary of the curved triangle of the center of the figure. For the purposes of this procedure it is needed to be able to find for any boundary of a subshape its coincident one in other subshape.

Boundaries also have a sense of alignment with others. Two boundaries of different subshapes are aligned if they *a)* are connected to a common intersection vertex and *b)* the end of one boundary is contiguous to the start of the other in the original shape (the single shape we computed division on). This means that you could travel from points at the end of one boundary to points at the start of the other without traveling any other points in between if you were drawing with a pen following the commands (line to and move to) that define the original shape, in order.

Each intersection point is connected to four boundaries, which are aligned two to two. Figure 4 shows the aligned boundaries connected to an intersection point of the example shape. Highlighted in the same color are the two pairs of aligned boundaries.

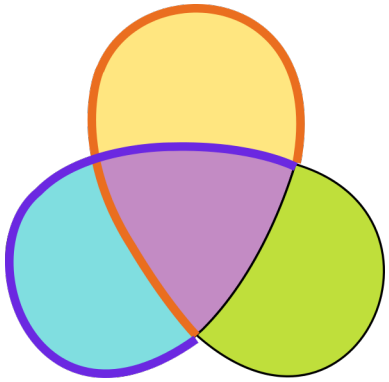

Figure 4: *Two aligned boundaries in blue and two aligned boundaries in orange.*

> **Issue 1:** I don't know any algorithm to get the result of the division operation plus all the required data mentioned in this step.

II. Step 2: Fill check

For each subshape it must be checked if the area inside corresponds to a fillable region in the original shape. This check is done even if the original shape is set by the user to not render any fill. The check will be made using the *evenodd* fill rule even if another value is set in the corresponding style property. `fill-rule` will affect only fill of the shape, not stroke.

Because subshapes cannot self-intersect by definition, checking if the subshape corresponds to a filled region is equivalent to checking if *any* point within the subshape is filled in the original shape.

We well refer to those subshapes that correspond to a fillable region as *fillable subshapes* and those which do not as *hollow subshapes*.

> **Issue 2:** How to check in Inkscape code if a point is within the fillable area of a shape?

Issue 3: How to get a point inside a shape avoiding degenerate cases? Raycasting at half the height of the shape and picking the median of the first interval may be a way if we can assure that there will not be a case where a subshape contains a zero-width triangle.

III. Step 3: Outline and stroke

The rest of the process depends on the desired stroke alignment.

III.1 Step 3a: Inset stroking

For inset stroking only fillable subshapes are considered.

The orientation of each subshape must be calculated (clockwise or counter-clockwise). If the subshape is clockwise, the subshape must be rendered with left stroke alignment. Otherwise, right stroke alignment must be performed.

In any case, the stroke must be clipped to the subshape so thick strokes do not cross the subshape boundaries.

Figure 5 shows the example shape with stroke inset.

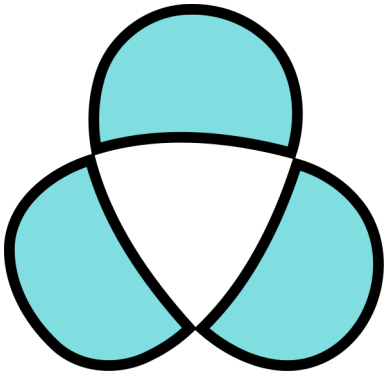

Figure 5: *stroke-alignment: inset*

III.2 Step 3b: Outset stroking

In order to accommodate self-intersecting shapes, outset stroking is somewhat more complex than inset stroking. For reference of the desired result, check both fig. 6, which shows the result of applying outset stroke to our previous example shape and fig. 7, which shows the expected result of the same operation over a shape with many more intersections.

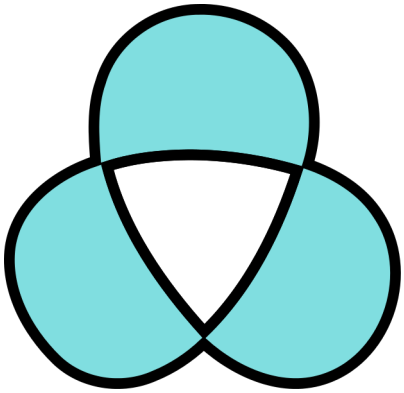

Figure 6: *Outset alignment on our first example shape.*

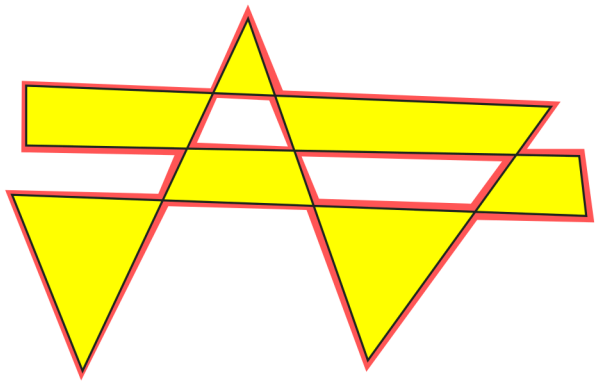

Figure 7: *Outset alignment on a complex polygon. The red line indicates the computed stroke. It is hand drawn, so the line width is not accurate.*

Let G be a graph-like structure. Each node represents an intersection point. Each node has exactly four connection slots, each holding an end of a boundary, represented as a graph edge. Coincident boundaries will be represented only in this graph.

Each connection slot has one of the four names in the vector $N = [''A_1'', ''B_1'', ''A_2'', ''B_2'']$. The connections are sorted by the angle of the unit vector with origin in the intersection vertex and direction

the tangent of the boundary end. In clockwise order, A_1 comes before B_1, that comes before A_2 and so on. The sorting of angles is relative, so there are four sets of equally valid sorts for each set of tangent vectors. Any of these will work.

Figure 8 shows a very simple graph for a figure with one intersection that could have the shape of the eight number. Notice that aligned boundaries share a letter, e.g. A_1 is aligned with A_2. Also, due to the angle sorting it is very easy to traverse boundaries with a defined direction: given a boundary connected to a certain slot of a node you can get the boundary to the right by traversing through the connection slot with the next name in the vector N, warping if necessary.

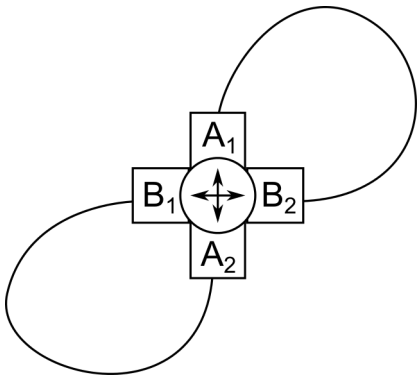

Figure 8: *The graph describing what could be an eight shape. The boundaries are drawn in a way that recalls this shape.*

Figure 9 shows the graph for a more complex shape which is self-intersecting.

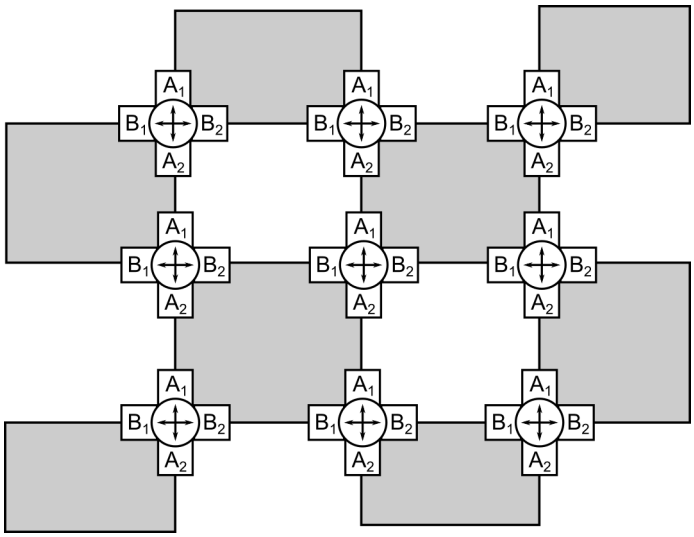

Figure 9: *The graph of a self-intersecting path. The colored areas are fillable subshapes.*

Having this graph with all the boundaries and intersection points, we remove the boundaries corresponding to hollow subshapes. These subshapes will be stroked inset as explained in section III.1.

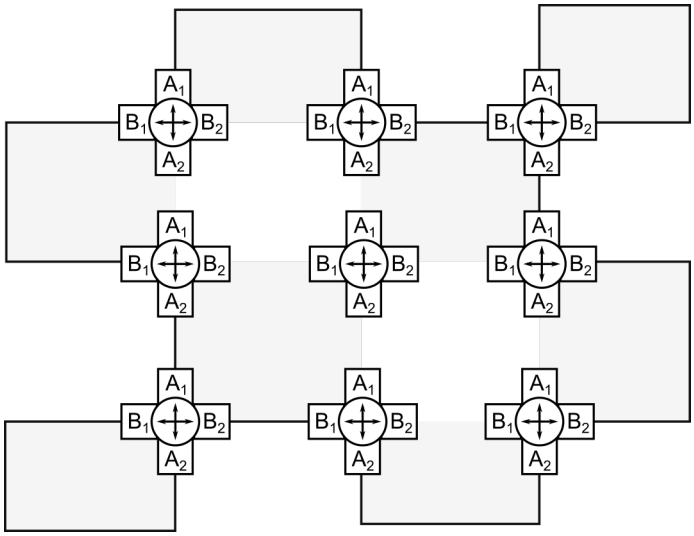

Figure 10: *The boundaries corresponding to hollow subshapes have been removed, greatly reducing the complexity of the graph, which now only contains the outer border of the shape.*

Figure 10 shows the graph for the previous example shape after removing boundaries of

hollow subshapes. Using this graph we fill find the outer border of the shape, which is defined as a path such that:

1. It traverses all the remaining boundaries reachable from the starting node.

2. No boundary is traversed twice.

3. Crossing a node always triggers a direction change, i.e. if the path arrives at a node at an *A* slot it must exit by a *B* slot and vice versa.

A simple algorithm for finding such a path consists on parting from any boundary of any selected start node in a given direction and gearing right at every intersection node found in the path until arriving the start node. Then do the same again, but gearing always left instead. One exploration will traverse the outside border of the shape while the other will traverse only a cycle in the graph or may try to gear towards a removed boundary.

Pick the path with the most boundaries as the outside border of the shape. If there are still boundaries to be traversed in the graph then the shape is disconnected. Remove the boundaries used by the picked path and repeat the algorithm starting from other node as many times as needed to find every outer border path.

Finally, the orientation of each border path must be calculated. Clockwise paths will be stroked with left alignment and counterclockwise paths will be stroked with right stroke alignment. The result for the previous example shape is shown in fig. 11.

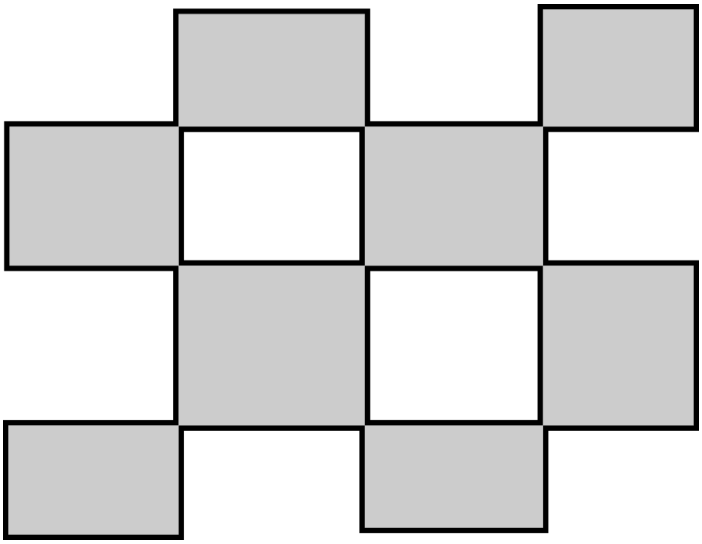

Figure 11: *The expected result of stroking a self-intersecting shape with outset alignment.*