Algorithms

Richard Dodd

June 19, 2017

Abstract

This is just a description of the algorithms used in plotter.
TODO floating point stability

1 Line Algorithm

The line algorithm *input* is a ordered list of points (x, y) where $x, y \in [-1, 1]$, and the line width w with $0 < w << 1$ (w should be much less than 1, a value like 0.01 may be satisfactory). If points are outside this range, they need to be scaled before this algorithm is used. The *output* is a vector of points, forming quadrilaterals as follows: the number of points will be $2n | n \in \mathbb{N}_0$, and the quadrilaterals are $(p_{2k}, p_{2k+1}, p_{2k+2}, p_{2k+3}) | k \in \mathbb{N}_0, 2k + 3 \leq n$, for example $(p_1, p_2, p_3, p_4), (p_3, p_4, p_5, p_6) \ldots$.

No points and one point are special cases that both produce no quadrilaterals. A single point can be shown with a circle of radius w (width), but that is not handled in this algorithm.

1.1 Expected output

Here I detail what properties the output should have:

- All points should be joined by lines.
- Lines should have even thickness.
- There should be a maximum of one quad per point (above the first)
- Asymptotic blow-up for parallel lines should be mitigated. (This may mean more than one quad because of bevelling)

These kinds of lines are already well handled by the SVG format. It makes sense to use a similar algorithm to SVG, except that we hard-code to use miter style linejoins [1] with a limit [2] of about 20 degrees, at which point we bevel.

We should be able to use the SVG algorithm as a guide, but we want to make sure our algorithm has the following properties:

- Minimal clock cycles, max speed.

- Minimal memory footprint, compact representation.
- CPU cache friendly (use the stack, heap use should be sequential and local).
- Any allocation happens once, no expensive `Vec` resizes.

Rust iterators should help us out here.

1.2 Derivation and Algorithm

First we derive a formula for the points of our surface, then we implement this efficiently. Each point from the input will have 2-3 points in the output (3 if we need to bevel). We handle the end point formula first since this is a special case and simpler than the interior point formula.

1.2.1 End Points

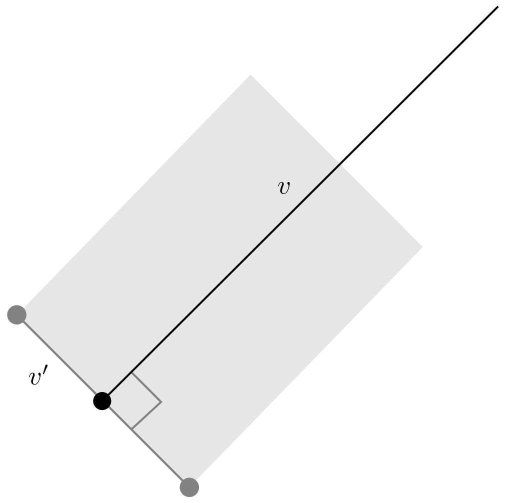

Figure 1: Visual representation of end point render

Using the definitions of v and v' in figure 1, we can derive v' from v, and then calculate the coordinates of the point.

We know that $v' \cdot v = 0$ and $v' \cdot v' = w^2$ where w is again width. This is equivalent to

$$v_y v'_y + v_x v'_x = 0 \qquad v'_y = -\frac{v_x}{v_y} v'_x \qquad v'_x = -\frac{v_y}{v_x} v'_y \tag{1}$$

and

$$(v'_y)^2 + (v'_x)^2 = w^2 \tag{2}$$

which gives

$$\left(\frac{v_x}{v_y}\right)^2 (v'_x)^2 + (v'_x)^2 = w^2 \qquad \left(\frac{v_y}{v_x}\right)^2 (v'_y)^2 + (v'_y)^2 = w^2$$

which after some rearranging gives

$$v' = (v'_x, v'_y) = \left(\pm w \frac{v_y}{|v|}\ ,\ \mp w \frac{v_x}{|v|}\right) \tag{3}$$

. The two solutions correspond to the two points we need. It is important to note that the two values have different signs - if the signs of v_y and v_x differ (i.e. negative gradient) then the sign of v'_x and v'_y will be the same.

Now we substitute our vectors for points: lets label the start of v as v_s, the end of v as v_e, and the ends of v' as v'_e (this represents 2 points). Then $v = v_e - v_s$ and $v' = v'_e - v_s$. Plugging into the above equation gives

$$(v'_{ex}, v'_{ey}) = \left(v_{sx} \pm w \frac{v_{ey} - v_{sy}}{|v|}\ ,\ v_{sy} \mp w \frac{v_{ex} - v_{sx}}{|v|}\right) \tag{4}$$

which is our formula for the line points. Here is the formula in python:

```python
def line_points(line, width=0.01):
    vsx = line[0][0]
    vsy = line[0][1]
    vex = line[1][0]
    vey = line[1][1]
    vlen = sqrt((vex - vsx) ** 2 + (vey - vsy) ** 2)

    vyn = (vey - vsy) / vlen
    vxn = (vex - vsx) / vlen
    return [[vsx + width * vyn, vsy - width * vxn],
        [vsx - width * vyn, vsy + width * vxn]]
```

and figure 2 is a graph of the resulting line thickness.

1.2.2 Interior Points

Now we need to derive a formula for interior points. These points depend on the point itself and one point either side of it. Figure 3 shows how we want to construct the line. See this figure for our labelling, noting that the subscripts are chosen so 0 is the point before, and 1 is the current point. We want to find the co-ordinates of p_i, the point where the 2 exterior lines meet.

We will exploit the normals at the points as marked to calculate the two points we will draw to. The procedure at a high level is

- Calculate q_0 and q_1 using the fact that $\boldsymbol{v}_0 \cdot \boldsymbol{v}_p = 0$ and $\boldsymbol{v}_0 \cdot \boldsymbol{v}_0 = |\boldsymbol{v}_0|^2 = w^2$, with similar formulae for the next point p_2.

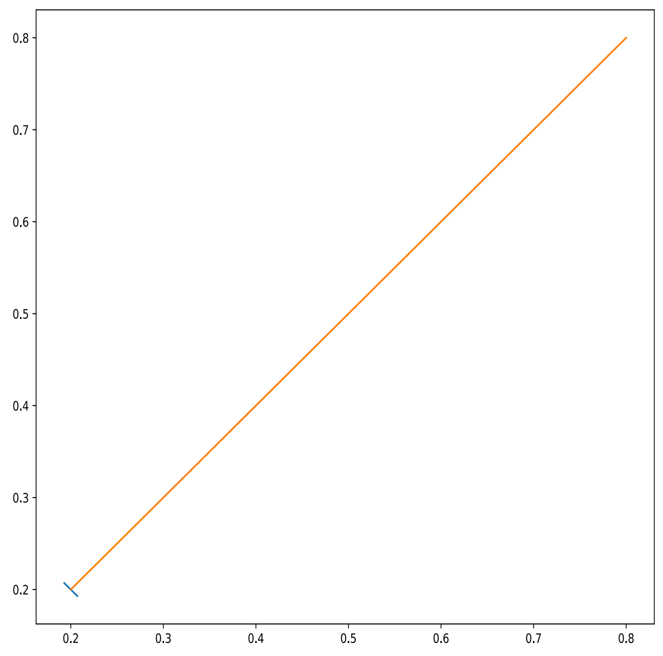

Figure 2: Graph of python implementation (Line is $(0.2, 0.2) \rightarrow (0.8, 0.8)$, width is 0.01)

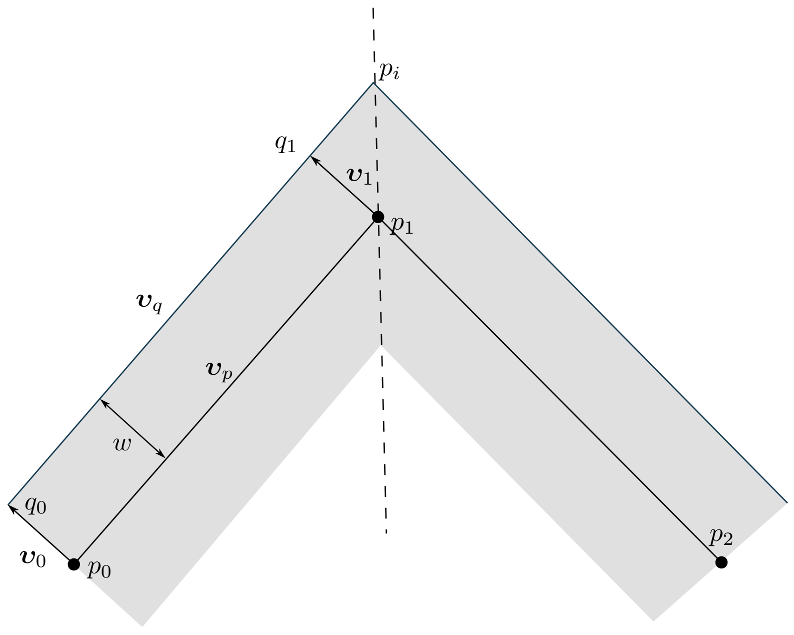

Figure 3: An interior point

- Use these points to give y-intercept[1] for the boundary line, handling a vertical line with no y-intercept.
- Do the same for the point after the current point (p_2, not shown).
- Find the point where the lines meet (note special case where lines are parallel).
- Convert mitre to bevel if necessary (if the point is too far from p_1)[2].
- Compose all these formulas and see how to minimize work done (cancel where possible etc.)

For the derivation, let $\boldsymbol{v}_a$ and p_a be defined as in figure 3 for various a, and define $\boldsymbol{v}_{ax}$ and p_{ax} as the x components of $\boldsymbol{v}$ and p, with y components defined similarly. Also define $m = \frac{\boldsymbol{v}_{py}}{\boldsymbol{v}_{px}}$ as the gradient of $\boldsymbol{v}_p$. Then the following are true:

$$\boldsymbol{v}_p \cdot \boldsymbol{v}_0 = 0 \qquad \boldsymbol{v}_p \cdot \boldsymbol{v}_1 = 0 \tag{5}$$

$$\boldsymbol{v}_0 \cdot \boldsymbol{v}_0 = |\boldsymbol{v}_0|^2 = w^2 \qquad \boldsymbol{v}_1 \cdot \boldsymbol{v}_1 = |\boldsymbol{v}_1|^2 = w^2 \tag{6}$$

[1]Gradient is the same as for the line itself.

[2]We will skip this step for now, but when we include it we may have 2 data points rather than 1 for this point, and this will need to be considered during triangulation.

Taking the $\boldsymbol{v}_0$ case in equation 5 ($\boldsymbol{v}_1$ is similar),

$$\boldsymbol{v}_{0x}\boldsymbol{v}_{px} + \boldsymbol{v}_{0y}\boldsymbol{v}_{py} = 0$$

so

$$\boldsymbol{v}_{0x} + m\boldsymbol{v}_{0y} = 0 \qquad \boldsymbol{v}_{0y} + m^{-1}\boldsymbol{v}_{0x} = 0 \qquad \boldsymbol{v}_{0x}^2 + \boldsymbol{v}_{0y}^2 = w^2 \tag{7}$$

and solving for $\boldsymbol{v}_{0x}$ and $\boldsymbol{v}_{0y}$ gives

$$\boldsymbol{v}_{0y}^2 = \frac{w^2}{m^2+1} \qquad \boldsymbol{v}_{0x}^2 = \frac{w^2}{m^{-2}+1} \tag{8}$$

so

$$q_{0y} = p_{0y} \pm \frac{w}{\sqrt{m^2+1}} \qquad q_{0x} = p_{0x} \pm \frac{w}{\sqrt{m^{-2}+1}} \tag{9}$$

. Note that there are always 2 solutions; this corresponds to both sides of the line (although we only marked one of them on the figure). Now that we know a point on the line, and the gradient, we have an equation for the line:

$$y - p_{0y} \mp \frac{w}{\sqrt{m_0^2+1}} = m_0 x - m_0 p_{0x} \mp \frac{m_0 w}{\sqrt{m_0^{-2}+1}} \tag{10}$$

where we have labelled m as m_0. Now we can repeat this for the point after (labelled p_2). Let's relabel the m above as m_0, and the m for p_2 as m_1. Then we also have

$$y - p_{1y} \mp \frac{w}{\sqrt{m_1^2+1}} = m_1 x - m_1 p_{1x} \mp \frac{m_1 w}{\sqrt{m_1^{-2}+1}} \tag{11}$$

and the point the lines intercept (p_i) is given by solving (10) and (11), giving

$$(m_1 - m_0)p_{ix} = m_0 p_{0x} - m_1 p_{1x} \pm \frac{m_0 w}{\sqrt{m_0^{-2}+1}} \mp \frac{m_1 w}{\sqrt{m_1^{-2}+1}} \tag{12}$$

$$+ p_{1y} - p_{0y} \pm \frac{w}{\sqrt{m_1^2+1}} \mp \frac{w}{\sqrt{m_0^2+1}} \tag{13}$$

for p_{ix}, and

References

[1] `https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-linejoin`

[2] `https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/stroke-miterlimit`