Algorithms

Richard Dodd

May 30, 2017

Abstract

This is just a description of the algorithms used in plotter.

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.

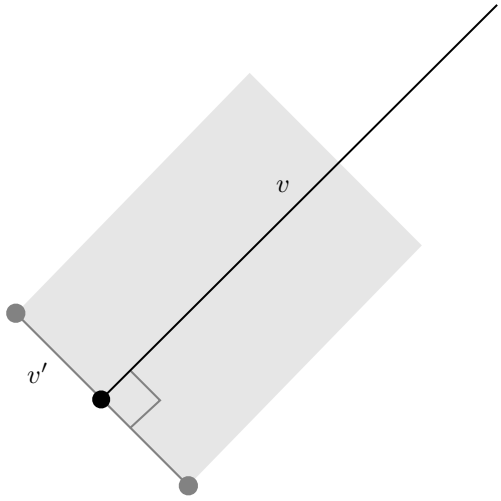

Figure 1: Visual representation of end point render

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

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

and

$$(v'_y)^2 + (v'_x)^2 = w^2$$

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

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

. 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.

References

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

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