Fast method for rendering vector graphics with Green's theorem

Cristhian Grundmann

June 15, 2023

Mathematical definition of rendering

Figure 1 shows a polygonal path in red to be filled. For simplicity, consider that the rendering process is all about assigning a real number $v \in [0, 1]$ for each pixel.

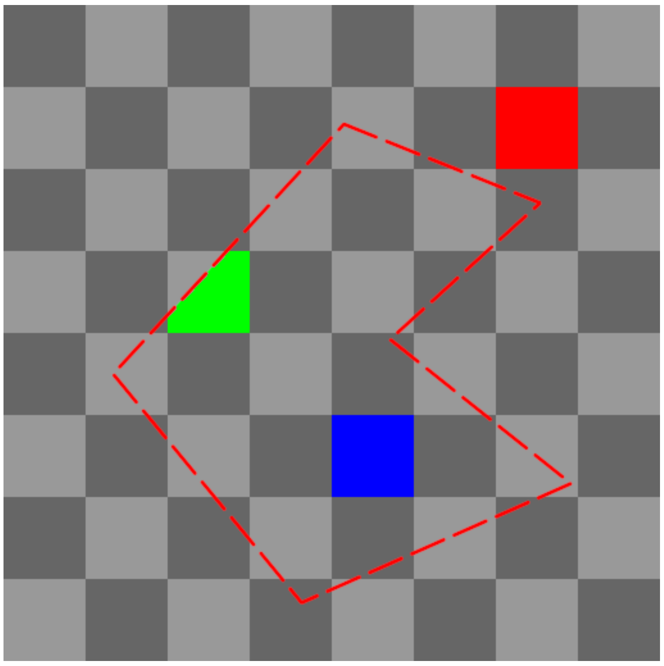

Figure 1: Polygonal path to be rendered

The pixels are 1 by 1 squares and also have area 1. The number v is defined as the intersection area of the region with the pixel.

Since the blue pixel is completely inside the region, it has $v = 1$. The red pixel has $v = 0$ because it's 100% outside the region. The green pixel is partially covered, and it has some $v > 0.5$

Computing v for each pixel decides which ones are outside and inside the region and also solves the aliasing problem. On a simple optical reasoning, the area of intersection can be interpreted as the amount of light detected by the pixel's sensor.

For simplicity, consider the path to be always polygonal.

Formula for computing areas

The **projection** of a point p onto a pixel of region S is defined as the point of S closest to p. If p is inside the pixel or in it's border, than the projected point is p itself. Otherwise, the projection is in the pixel's border.

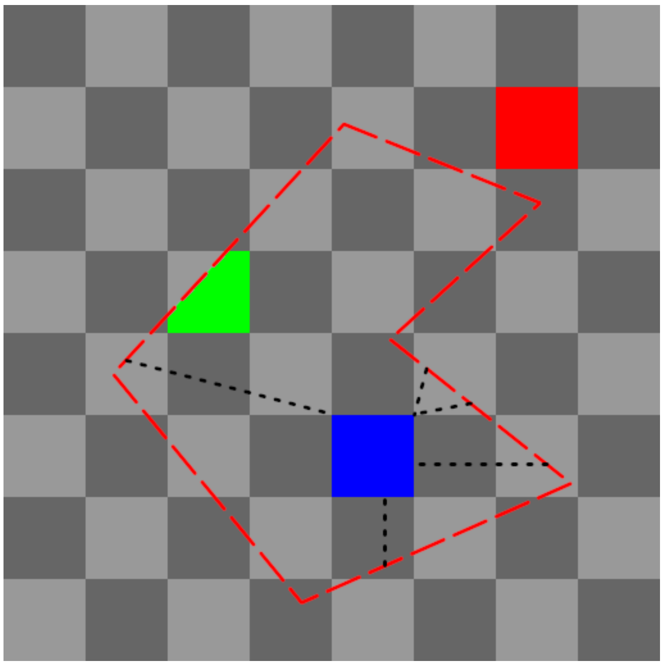

Figure 2: Projection of some points onto the blue pixel

The area of intersection can be computed as the area of the projection of the polygon path onto the pixel.

Figure 2 shows that the projection of the path covers all points on the blue pixel.

The projection of a path is also a path. This means that computing the intersection area is the same as computing the area of the projected path. Note that some piece of path can be projected onto a single corner on the pixel, and also that the projection can move on the pixel's border and come back, undoing the previous motion. This can be seen in Figure 2: the path's projection above the blue pixel briefly moves back and forth, canceling it's own "effect".

Finally, the signed area can be computed using the Green's theorem applied to computing areas:

$$A = \oint_C x \, dy$$

Where C is the oriented path of interest. The path's orientation determines the sign of the signed area. Anticlockwise orientation yields a positive area, and clockwise orientation yields a negative area. When drawing the letter **O**, for instance, the outside round path must be counterclockwise, and the inside path must be clockwise. This means that the inner region subtracts area from the outer region, creating the hole.

Since the area is an integral over the path, it's possible partition the path, compute the individual integrals and sum them all to get the area.

It's very convenient to partition the path based on the pixels. That is, every time the path crosses a pixel border, a cut is made, making each piece on a single pixel, like the pieces on Figure 3.

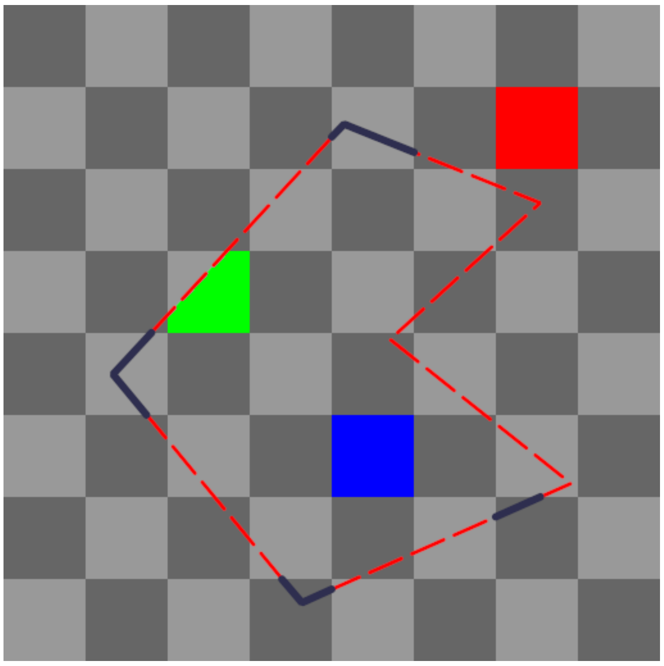

Figure 3: Some pieces of the path

If a piece is more than a line segment, break it down even further in multiple line segments.

Consider a fixed pixel P to compute its coverage area. Consider also that the coordinate system's origin is at the lower left corner of P. For every path piece, an integral must be computed.

If the piece is in P itself, its projection is itself, and the integral becomes $\frac{x_0+x_1}{2}(y_1 - y_0)$ for each segment from (x_0, y_0) to (x_1, y_1) inside the piece.

If the piece is in a pixel of a different row than P, then the projection is either on a fixed corner or completely horizontal, making $dy = 0$, thus the integral becomes 0.

If the piece is in the same row but to the left of P, then the projection is always on the vertical line of $x = 0$, thus the integral also becomes 0.

Finally, if the piece is on the right of P, then the intersection is always on vertical line of $x = 1$, and the integral becomes $y_1 - y_0$ for every segment of the piece.

Figure 4 illustrate these possibilities.

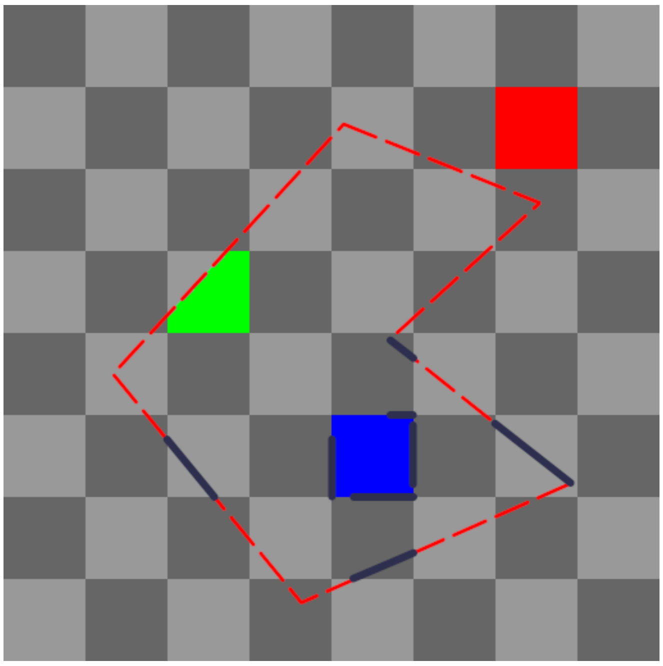

Figure 4: Some pieces and their projections

On the opposite perspective, consider a single line segment piece of the path, from (x_0, y_0) to (x_1, y_1). All the pixels affected from this piece are on the same row. The pixel from the piece itself gains an area of $\frac{x_0+x_1}{2}(y_1 - y_0)$. The pixels to its left gain an area of $y_1 - y_0$. These are all the pixels that can gain areas different than 0. Note that all the pixels to the left gain the same area. This provides an opportunity for optimization. All there is left to do now is to process all the line pieces, adding area to the corresponding pixels.

Figure 5 illustrates why the formula works.

The two pieces on the right have positive dy, so they're adding area to everything to its left. Both pieces on the left have negative dy, so they're subtracting area from everything to their left, canceling the extra area previously added.

Optimization

As mentioned on the previous chapter, all the pixels to the left of a piece gain the same area. This can be optimized.

Instead of having a 2-D array of v values, consider a 2-D array of δ values where $v_{ij} = \sum_{k=0}^{i} \delta_{kj}$. The array is initialized with 0 on all pixels.

With this array, adding a z to every v on a row j from index i and above is done by simply adding z to δ_{ij}.

Adding z starting from i_0 and ending in i_1 is done by adding z to $\delta_{i_0 j}$ and subtracting z from $\delta_{i_1+1,j}$. Note that this can be used to affect a v value of a single pixel.

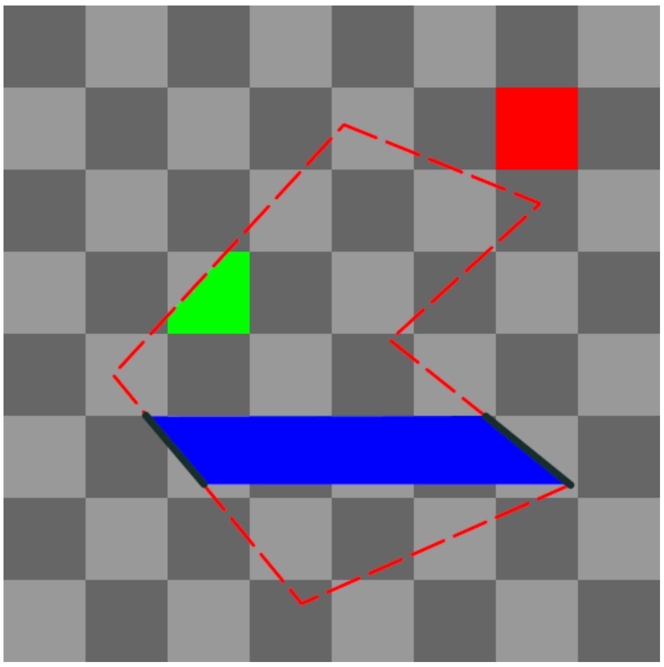
Figure 5: Part of the area computed using 4 pieces

This means that to process a line segment inside a pixel, only 3 changes need to be made in the δ array.

To add area to the pixels to the left:

$$\delta_{0j} \to \delta_{0j} + y_1 - y_0$$

To add area to the pixel bounding the current piece, and also stop the previous area:

$$\delta_{ij} \to \delta_{ij} + \frac{x_0 + x_1}{2}(y_1 - y_0) - (y_1 - y_0)$$

To stop the previous area:

$$\delta_{(i+1),j} \to \delta_{(i+1),j} - \frac{x_0 + x_1}{2}(y_1 - y_0)$$

To compute the v values, simply add the partial δ values and collect the results.

This optimization yields the following time complexity for the rendering process. The time taken to update the δ array is $O(s)$ where s is the total perimeter of the path, because only 3 writes are done for each piece inside a pixel. The time taken to obtain the v array is $O(WH)$, where W is the total width of the image, and H is the total height.

Note that the first write on the δ array, on column 0, always cancels out with another part of the path if it's closed and inside the frame. Another way of writing the buffer is to do the last 2 writes normally and change the first to a conditional one:

$$\delta_{0j} \to \delta_{0j} - (y_1 - y_0)$$

only when the piece has $x < 0$.

Now these writes have a slightly different geometrical interpretation. Instead of adding area to the left, they are subtracting area to the right. If a piece has $x < 0$, then it's necessary to write to δ_{0j}.

Conclusion

The v array can be used as a mask to apply colors from a source into a target surface. If $v = 0$, then the pixel is totally outside the current mask and the target pixel isn't changed. If $v = 1$, then the pixel is totally inside the current mask and the target pixel is changed or blended with the new source color. If v is a value between 0 and 1, then the target pixel is blended with the source pixel, where the weight of the source color is v. This solves the aliasing problem automatically.