Supplementary Materials for "VecFontSDF: Learning to Reconstruct and Synthesize High-quality Vector Fonts via Signed Distance Functions"

Zeqing Xia*, Bojun Xiong*, Zhouhui Lian†
Wangxuan Institute of Computer Technology, Peking University, China

1. Details of Post-processing Steps

1.1. From Parabolic Curves to a Shape Primitive

The raw input of VecFontSDF consists of the parameters of parabolic curves. As mentioned in the main manuscript, each parabolic curve is defined by:

$$k(px+qy)^2 + dx + ey + f = 0, \tag{1}$$

and the inside area of the parabolic curve is defined as:

$$H(x,y) = k(px+qy)^2 + dx + ey + f < 0. \tag{2}$$

To reconstruct the geometry of an input glyph image, we first need to compute the intersection of N_a areas to get a shape primitive. To realize this, we start from a initial square canvas from the left-bottom point (-1,-1) to the right-top point (1,1). The four sides of this initial square canvas can be treated as four special quadratic Bézier curves which degenerate into straight lines.

As shown in Fig 1, when adding a new parabolic curve (one of the N_a parabolic curves), we need to calculate the intersection of the inside area of the newly-added parabolic curve and current canvas. The current canvas (the red area) is enclosed by several quadratic Bézier curves and the inside area of newly-added parabolic curve (the blue area) is depicted by Eq. 2. Therefore, the key problem to be resolved here is how to calculate the intersection points of a quadratic Bézier curve and a parabolic curve. Recall that, a standard representation for the quadratic Bézier curve can be described as:

$$\begin{aligned} P(t) &= (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \\ 0 &\le t \le 1. \end{aligned} \tag{3}$$

We simply substitute the x and y in Eq. 1 with the coordinates of the points in Eq. 3 to get a new quartic equation of t. After solving this quartic equation and validating $0 \le t \le 1$, we can obtain all the intersection points (the yellow points in Fig 1).

[1]Denotes equal contribution.
[2]Corresponding author. E-mail: lianzhouhui@pku.edu.cn

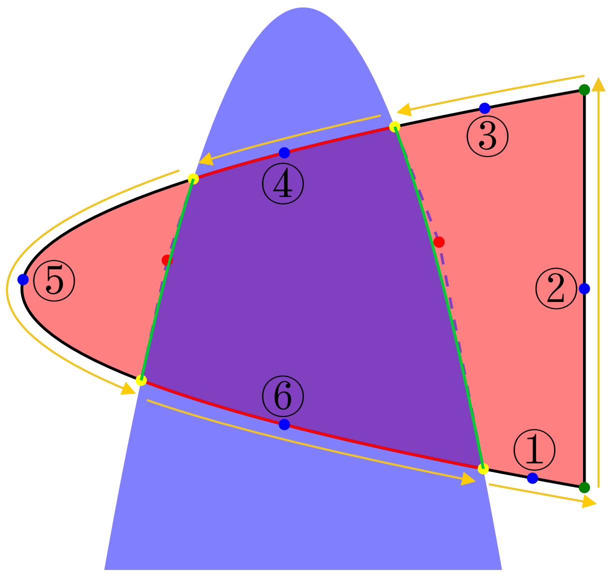

Figure 1. A detailed illustration of how to calculate the intersection of the current canvas and a newly-added parabolic curve.

Then, we need to update the current canvas (the red area) to the newly-calculated intersection region (the purple area). The newly-calculated intersection points (four yellow points) divide the current canvas into six segments as marked in Fig. 1. Due to the divisibility of Bézier curves, a given Bézier curve

$$B : P(t) = \sum_{i=0}^{n} \left[C_n^i t^i (1-t)^{n-i} P_i \right] \tag{4}$$

can be divided by any point $\hat{t}$ on the curve into two parts and every part is still a Bézier curve with the same order as the original one. For example, the left part B_l can be defined

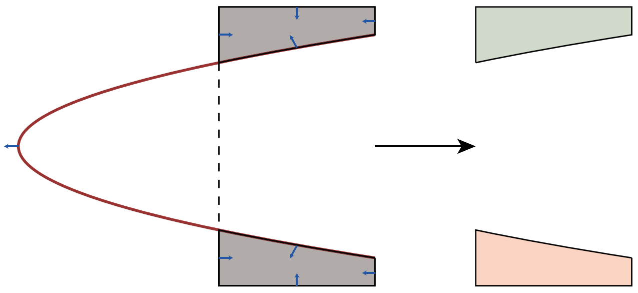
Figure 2. A typical corner case: a newly-added parabolic curve (the red curve on the left) splits the existing canvas into multiple parts (right). Blue arrows denote the directions of the inside areas from Eq. 2.

as:

$$B_l : P_l(t) = \sum_{i=0}^{n} \left[C_n^i t^i (1-t)^{n-i} \hat{P}_i \right]$$
$$\hat{P}_i = \sum_{j=0}^{i} \left[C_i^j \hat{t}^j (1-\hat{t})^{i-j} P_j \right], \tag{5}$$

and so is the right part. Therefore, these six segments are also quadratic Bézier curves.

To accurately obtain this intersection region (the purple area), we start from one of these intersection points (four yellow points), for example, the right-bottom yellow point. Then, we traverse anticlockwise through all the segments one by one (as pointed out by the yellow arrows in Fig. 1) and judge whether every segment is inside the parabolic curve (the blue area). Due to the convexity of quadratic Bézier curves, we only need to calculate if its middle point ($t = 0.5$) (the blue point) is inside this area. For the case shown in Fig. 1, the first three segments are not inside the blue area, so we simply ignore them and go to the right-top yellow point. The fourth segment (the red segment on the top) is exactly inside the blue area, so we reserve this segment and calculate the quadratic Bézier curve (a part of the parabolic curve that is marked as the green segment on the right) whose two on-curve control points are the current yellow point (right-top) and the previous yellow point (right-bottom).

Since the intersection point of the tangents of two on-curve control points of a quadratic Bézier curve is exactly the off-curve control point, we can calculate the equations of these two tangents (the purple dashed lines on the right in Fig. 1) to get the coordinate of the off-curve control point (the red point on the right). Based on Eq. 1, the slope y' of tangent on any point (x_0, y_0) can be calculated as:

$$2k\left(px_0 + qy_0\right)\left(p + qy'\right) + d + ey' = 0, \tag{6}$$

which equals to

$$(2k(px_0 + qy_0)q + e)y' = -(2k(px_0 + qy_0)p + d). \tag{7}$$

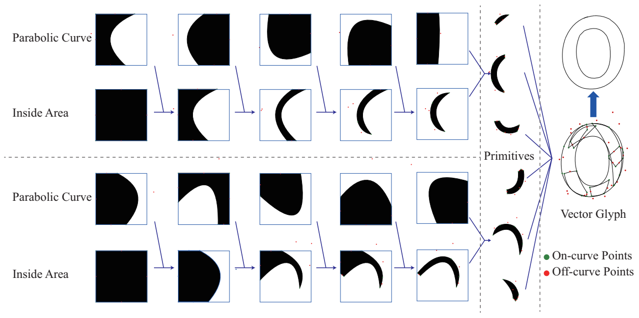

Figure 3. The pipeline of our post processing step (also shown as Fig. 4 in the main manuscript).

Therefore, the equation of the tangent on (x_0, y_0) is:

$$y - y_0 = y'(x - x_0), \tag{8}$$

where y' can be solved from Eq. 7.

After getting the two equations of tangents on two on-curve control points, we can use them to calculate the coordinate of the off-curve control point (the red point on the right). As shown in Fig. 1, we repeat the above process on the fifth and sixth segments where the sixth segment is inside the blue area to get another quadratic Bézier curve (the green segment on the left). Finally, we use these four quadratic Bézier curves (two red segments and two newly-calculated green segments) to update the current canvas.

We implement the above-mentioned process recursively, and eventually get the primitive as shown in Fig. 3.

It is notable that Fig. 2 shows a typical corner case, where a newly-added parabolic curve (the red curve in Fig. 2) splits the existing canvas (the gray area on the left) into several parts (the green and red areas on the right). Under this circumstance, we need to check whether the middle point of every segment from newly-added parabolic curve is out of the existing canvas. If so, we separate the existing canvas into the corresponding parts, and handle each part individually.

1.2. Calculating Intersection Points of Two Quadratic Bézier Curves

As shown in Fig. 3, after obtaining all the primitives, we can get the filled shape of the target vector glyph by simply assembling them together with many intersections inside the contour. However, further simplifying the primitives' outlines is necessary to obtain a complete outline representation of the vector glyph. Before we introduce the details of how to merge all the outlines of primitives, we need to clarify the process of calculating the intersection points of two quadratic Bézier curves which plays an important role in the following section.

To calculate all the possible intersection points of two quadratic Bézier curves described by Eq. 3, we have

$$\begin{aligned} \exists \quad & t_1, t_2 \in [0,1] \\ \text{s.t.} \quad & x_1(t_1) = x_2(t_2) \\ & y_1(t_1) = y_2(t_2). \end{aligned} \tag{9}$$

The left hand side of Eq. 9 equals to:

$$\begin{aligned} (1-t_1)^2 x_{1,0} + 2t_1(1-t_1)x_{1,1} + t_1^2 x_{1,2} = x_2(t_2) \\ (1-t_1)^2 y_{1,0} + 2t_1(1-t_1)y_{1,1} + t_1^2 y_{1,2} = y_2(t_2). \end{aligned} \tag{10}$$

We expand the above equation and let:

$$\begin{aligned} A &= x_{1,0} - 2x_{1,1} + x_{1,2} \\ B &= -2x_{1,0} + 2x_{1,1} \\ C &= x_{1,0} \\ D &= y_{1,0} - 2y_{1,1} + y_{1_2} \\ E &= -2y_{1,0} + 2y_{1,1} \\ F &= y_{1,0}. \end{aligned} \tag{11}$$

Thus, we can rewrite the Eq. 10 into:

$$At_1^2 + Bt_1 + C = x_2(t_2) \tag{12}$$

$$Dt_1^2 + Et_1 + F = y_2(t_2). \tag{13}$$

Let (12)$\times D-$ (13) $\times A$, we get

$$\begin{aligned} &(DB - AE)t_1 + (DC - AF) \\ =&Dx_2(t_2) - Ay_2(t_2). \end{aligned} \tag{14}$$

The left hand side of Eq.14 is a linear expression of t_1, and the right hand side is a quadratic expression of t_2.

Assume $DB - AE \neq 0$, we can represent t_1 by a quadratic function of t_2:

$$t_1 = \frac{Dx_2(t_2) - Ay_2(t_2) - DC + AF}{DB - AE}. \tag{15}$$

Then, we substitute t_1 in Eq. 10 with Eq. 15 and thus we can get a quartic equation of t_2. Afterwards, we solve this equation to get t_2 and use Eq. 15 to get t_1.

Otherwise, if $DB - AE = 0$, we directly solve the quadratic equation of t_2 from Eq. 14 and use this already known t_2 to solve one of the equations in Eq. 10 to get t_1.

After calculating t_1 and t_2 and validating if both of them satisfy the constraint $0 \leq t \leq 1$, we can finally use Eq. 3 to calculate the intersection points of these two quadratic Bézier curves.

1.3. Outline Simplification

As shown in Fig. 4, to merge the outlines of all primitives, we first need to calculate the intersection points (yellow points in Fig. 4) of all existing quadratic Bézier curves using the method described in Sec. 1.2. After calculating all the intersection points, every quadratic Bézier curve is further divided into shorter segments. Due to the divisibility of Bézier curves, these shorter segments are also quadratic Bézier curves.

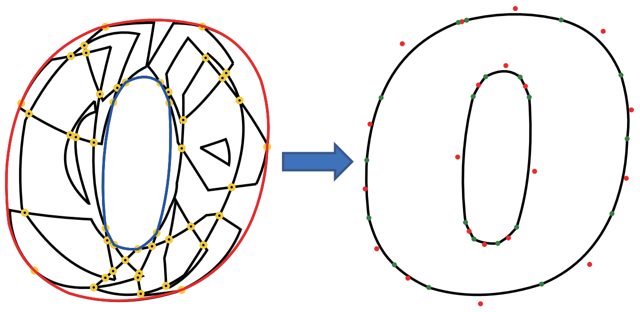

Figure 4. An illustration of our outline simplification step. Yellow points denote the intersection points of all quadratic Bézier curves. The red outline and blue outline compose this glyph's contour. Black segments are all other quadratic Bézier curves whose middle points' SDF values are less than 0 (i.e., inside the glyph contour).

For all the existing quadratic Bézier curves after division, we can directly ignore the ones inside the glyph contour (e.g., the black segments in Fig. 4) and reserve the ones composing the glyph contour (all the quadratic Bézier curves lying on the red and blue contours). Recall that in the main manuscript we have already get the pseudo distance functions $G(x, y)$ to calculate the value of the pseudo distance on each sampling point (x, y), which is an effective tool to help us determine the position of every segment. Due to the convexity of quadratic Bézier curves, whether a quadratic Bézier curve is contained in the glyph contour or not can be judged by the signed distance of its middle point ($t = 0.5$) $G(x_m, y_m)$, where (x_m, y_m) denotes the coordinates of the middle point.

For a given quadratic Bézier curve, if $G(x_m, y_m) = 0$ which means it composes the glyph contour, we reverse this curve and discard the curves whose $G(x_m, y_m) < 0$ (the black segments). Finally, we obtain all the quadratic Bézier curves lying on the glyph contour (the red and blue contours) to finish our outline simplification step.

2. More Results

In this section, we provide additional experimental results in support of the conclusions drawn in the main manuscript.

2.1. Vector Font Reconstruction

Fig. 5 shows more vector font reconstruction results obtained by our method as well as the corresponding glyph contours after implementing the simplification step mentioned in Sec. 1.3, from which we can see that our method

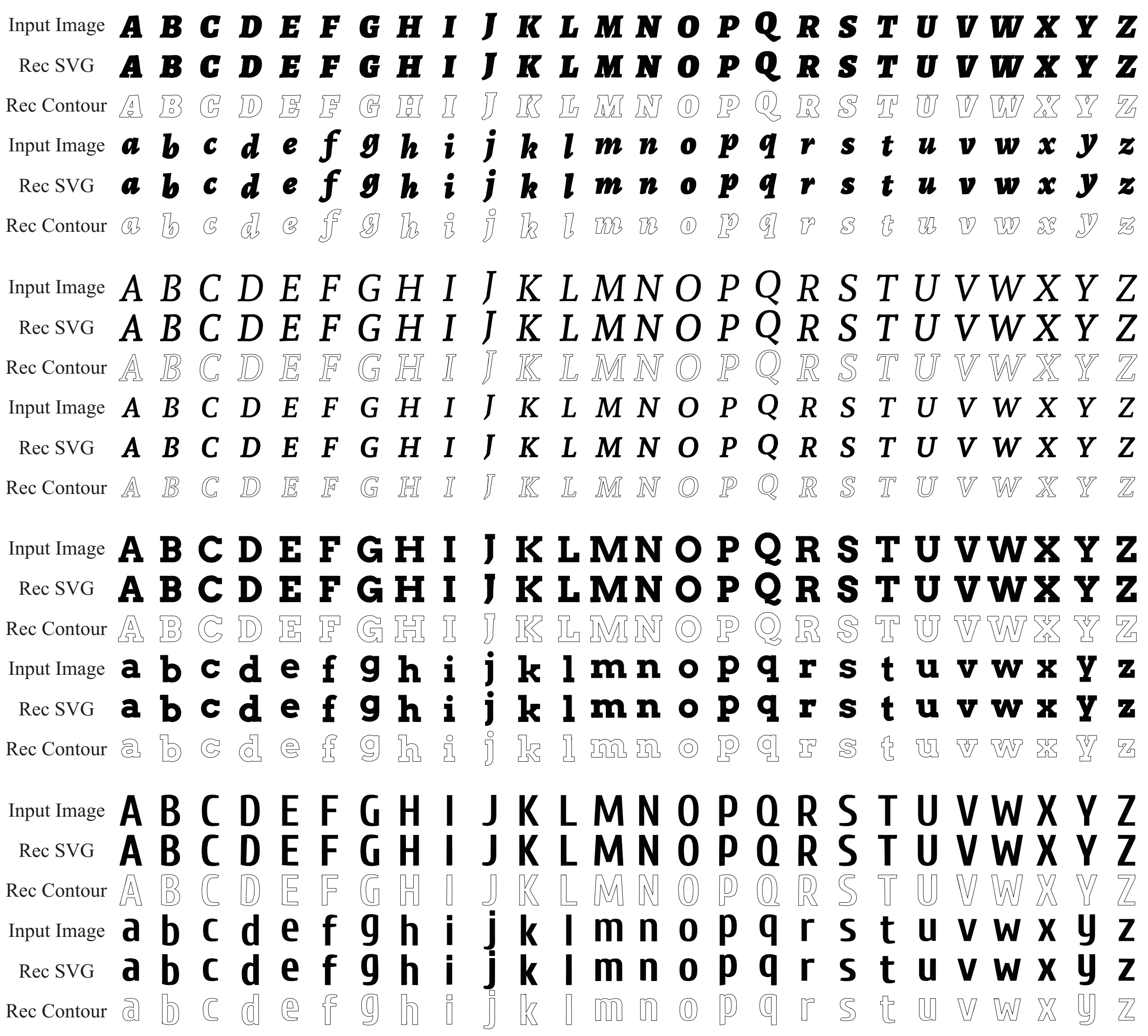

Figure 5. More vector font reconstruction results obtained by our method. “Rec SVG” denotes the filled shape of the reconstructed vector glyphs and “Rec Contour” denotes the contours of corresponding vector glyphs.

is capable of reconstructing various styles of fonts given the input raster images, including some complex glyphs with serifs.

2.2. Vector Font Interpolation

Fig. 6 shows more results on vector font interpolation by only inputting raster glyph images (denoted as the columns ‘g_1’, ‘g_2’ and ‘g_3’ in Fig. 6). Our method achieves smooth interpolation between different styles of fonts demonstrating that the latent space embedded by our CNN encoder is perceptually smooth and interpretable to represent the vector font style.

2.3. Few-shot Style Transfer

Fig. 7 shows more few-shot style transfer results generated by our method. As shown in Fig. 7, we demonstrate the effectiveness of our method in the task of generating all other vector glyphs by only giving a few reference glyph images instead of vector glyphs. The input reference glyphs are marked by red rectangles in Fig. 7, from which we can see that our model is capable of synthesizing the whole vector font given just a small number of reference glyph images. Especially for the second and third fonts, almost all glyphs in the synthesized vector font are approx-

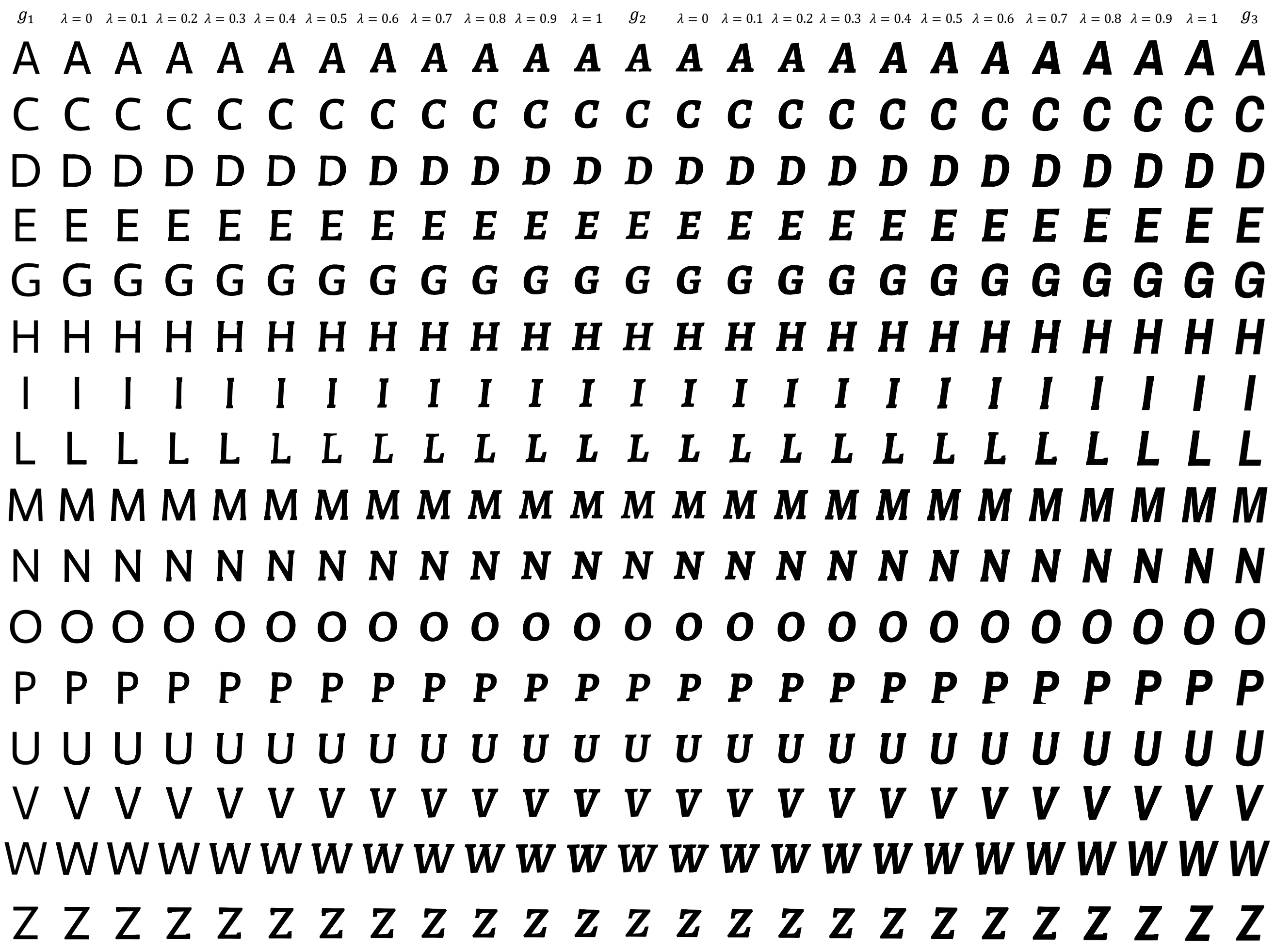

Figure 6. Our method is capable of generating vector fonts in new styles by only providing raster glyph images (the columns 'g_1', 'g_2' and 'g_3') in different font styles. Two interpolation processes ('g_1' to 'g_2' and 'g_2' to 'g_3') are presented in succession.

imately identical to the human-designed vector glyphs. For some complex and complicated fonts with serifs (such as the first and fourth fonts in Fig. 7), we observe that some local details of our synthesis results are only slightly different against the ground-truth glyphs. Considering that our model only receives the glyph images of 'A', 'B', 'a' and 'b' as reference inputs, our synthesized glyphs of other characters have already sufficiently embodied the style feature of these input samples. Most importantly, all the vector glyphs generated by our method are of high quality and look visually pleasing, which markedly outperform other existing state-of-the-art methods without further refinement.

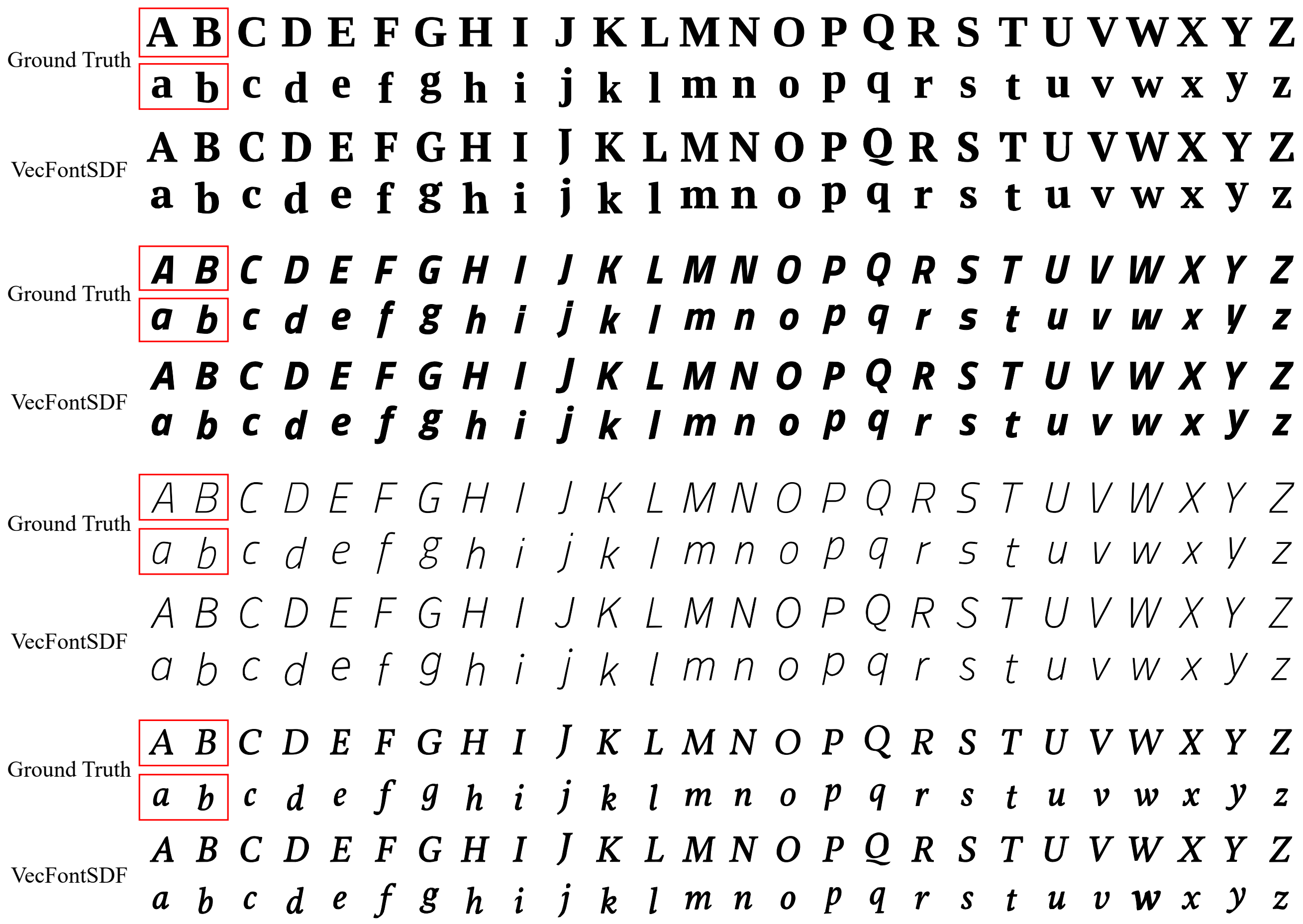

Figure 7. More few-shot vector font generation results of our VecFontSDF. The inputs to our networks are only the rendering results (raster images) of corresponding vector glyphs marked by red rectangles.