Learning 2d shape representations with Neural Networks

Ken Voskuil (2553740), Yannick Ligthart (1453580), and Mihai Ghidoveanu(2739259)

Abstract. Signed Distance Functions are a useful representation of geometry, that can modeled by neural networks with relative ease. We use SDFs to model font glyphs, and show that a single model can represent a large number of glyphs through a learned shape embedding. By manipulating the learned embeddings, we can generate new, unseen fonts.

Keywords: SDF · Neural Networks · Fonts · PCG

1 Introduction

Procedural content generation (PCG) for games is an active field of research, both in academic circles and the game industry. Applications range from tools for helping game designers to generating complete game worlds and even rules autonomously. Recent developments around generative neural network models spark a lot of interest in using these machine learning techniques in the context of PCG. Many of the recent works focus on generating bitmap or bitmap-like data. Some are dedicated to generating geometry more directly in the form of meshes or point clouds. In our work we explore another alternative, using neural networks to model geometry using a continuous implicit representation. Signed distance fields, or SDFs, describe geometry with a mapping from points to the shortest distance to the surface of the geometry. Although SDFs have been long know in the game industry, they were only introduced very recently in the context of machine learning.

We will start with a short discussion of SDFs to establish the context of our work. We follow with our method for representing SDF functions using neural networks and present some results. We compare the effectiveness of different shape representations, and explore the possibilities that a learned representation provides us. We try to generate new shapes by exploring this learned representation. We conclude with some potential future research directions and a conclusion.

2 Signed Distance Functions

There are many ways to represent geometry, including point clouds, voxel or bitmap grids, vector paths or meshes, parametric functions, or implicit functions. We focus on a specific type of implicit representation, known as Signed Distance

Functions, or SDFs. These functions map 2D or 3D coordinates to a real number. A shape is represented by a mapping from a point to the shortest distance to the surface of said shape. The sign is used to indicate if the point lies inside or outside the shape, with the common convention that the points inside the shape map to negative values. By rendering the surface where the function is equal to 0, we will get a visual representation of the object.

Throughout this report, we will use the notation SDF_x to refer to the signed distance function describing a shape x. A simple example of an SDF is one that describes a circle centered at some point $\boldsymbol{p}$ with a radius r. For any point $\boldsymbol{v}$, we have:

$$SDF_{circle}(\boldsymbol{v}) = ||\boldsymbol{p} - \boldsymbol{v}|| - r$$

SDFs have interesting characteristics that make them useful in different applications. Multiple SDFs can be combined using a rich vocabulary of operators, including the union and intersection of two shapes. This makes them a popular tool for Constructive Solid Geometry. SDFs are also well known in the demo scene [1], where high fidelity graphics are delivered in space restricted programs of 4 to 64 kilobyte. Intricate shapes can be constructed with few lines of code, and rendered real time using sphere tracing. Another well known application of SDFs is for font rendering in games. If rendering fonts is too costly, the font can be rendered to a bitmap offline. This introduces a quality trade-off: larger bitmaps result in higher image quality, but also costs more memory and more computation time to render. Green [7] found that rendering the SDF of a font to a bitmap yields much better results, especially when scaling. For a more in-depth review of SDFs, we refer to this blog [15].

Our interest to use SDFs with neural networks comes from the ability to represent any shape with a simple function. As neural networks are universal function approximators, any SDF, and thus any shape, can be (approximately) represented by a neural network. Taking the SDF for a circle as an example, we can train a neural network that takes a two dimensional input and produces a single output on a set of points $X \subset \mathbb{R}^2$ and labels $y = \{SDF_{circle}(x) : x \in X\}$.

Our goal is to extend this idea to train a model that can generate new shapes. We train a neural network on multiple shapes in different classes, by introducing an encoding for each shape. With this trained model, we should be able to generate new shapes that fit in the classes of the original data, by exploring the shape encodings as a latent space. This goal resides in the topic of representation learning and there have been various previous works on this.

3 Related Work

Using deep learning in representing complex shapes has lately been a research topic in various forms. Initially, neural networks have been used on voxel and mesh representations [8, 16], and recently, on SDF representations [14] and the closely related occupancy functions [5, 12] or level sets [13]. Learning an SDF representation, compared to a mesh / voxel, has the advantage it does not need

a specified resolution. The training data is sampled from a continuous function, and the resulting model should be able to learn as much detail as it can represent. Furthermore, the neural networks has been proven to learn better from a continuous SDF representation than a discrete level set binary representation [9], mainly due to the additional global information the distance function brings.

A disadvantage to using SDFs representation is the big number of queries needed to render the geometry. This leads to high computational costs when using the geometry represented by a neural network, but recent works are working on creating a differentiable, more efficient renderer to address this issues [11].

Deep learning representation using SDFs have already started being used in different applications, ranging from aerodynamics simulation [9] to real-time feet tracking [6]. Our work will tackle 2D font generation. While usual CNNs or GANs can be used to learn or generate bitmap representations [10], we pursue learning SDFs representation to access infinite-resolution capacity as well as gaining from the richer representation the SDF provides about the shape's implicit geometry.

4 Method

We start our work by confirming our premise, that neural networks can effectively model SDFs, by training neural networks on a single shape. We build on this by training on multiple shapes, followed by multiple classes of shapes, and finally by introducing an explorable latent space. The methodology for each step is described below, after a discussion on our dataset.

4.1 Creating the dataset

For our training data we need a collection of shapes, represented as an SDF. We use a selection of fonts for this purpose, motivated by the following properties:

1. There are many fonts available for free under open source licenses.
2. Glyph shapes are complex, and can be classified both by which character a glyph represents and the font it belongs to. The latter can also be nuanced further using the stylistic characteristics of the font.
3. We can extract SDFs from fonts using existing code. Font files contain glyphs that represent each character. These shapes are typically described by cubic bezier paths. The STB_truetype C library [4] parses truetype font files and includes functionality to transform the path defining a glyph to an SDF. With small modifications, we were able to expose this function to our python program in the way that we needed for this project.

We use the following fonts in our project:

- Times New Roman
- Junicode
- EB Garamond
- Libre Baskerville

- Work Sans
- Oswald
- Archivo

Except for Times New Roman, which is included with Windows, these fonts were sourced from [2] and are available under an open source license. While font data follows a very specific format, SDFs are able to represent any geometry, and our methods translate directly to source data other than fonts.

When we have the SDF of a shape, generating labeled training data is a two step process. First, we sample points from a normal distribution as the input, and then we apply the SDF to each point to get the label. We found it is important to choose a good point distribution. The area near the surface of the shape is the hardest to model correctly, so you would want to focus your data points there. We found that choosing a standard deviation of two to three times larger than the bounding radius of the shape you are trying to model worked well. For our dataset this means we use $\sigma = 0.3$.

4.2 Neural SDFs

Our first goal is to show that a neural network can effectively model an SDF, by training it on a single shape. We use the SDF of the glyph for lowercase `a` from *Times New Roman*.

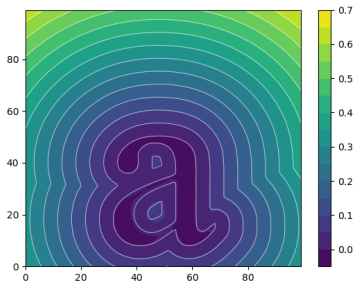
We trained a fully connected network to regress on the distance of an input point to the boundary. We chose to use three layers, each with 512 hidden units and using ReLU activation. The output layer has just a single node without activation. We found that when we use the mean square error as the loss function, the models performed reasonably well, but has trouble representing the finer details of the `a`. Especially in parts where the shape is thin we see gaps in the shape. To improve this, we designed a custom loss function, that magnifies the error near the surface. We achieve this by scaling the error by the label value y. The function takes the form of

$$\mathcal{L}(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{n}\sum_{i=1}^{n}(\frac{y_i - y_i'}{max(abs(y_i), \epsilon)^c})^2$$

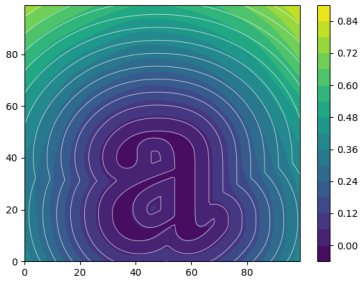
Where c is a constant to control how much we magnify the error near 0, and ϵ serves as a magnification limit. We use $c = 0.2$ and $\epsilon = 0.0001$ for our experiments. The intuition that lead us to this loss function is that the value of an SDF changes much more rapidly near the surface than far away from it. For many applications, the most important part to get right when approximating an SDF is where the function equals 0, because this is where the surface is. When converting an SDF into an explicit geometry representation, such as a mesh, predicting 100.1 instead of 100 might not matter, while predicting 0.1 instead of 0.001 could move the surface of resulting mesh, depending on the sensitivity of the discretisation.

Figure 1 shows the results of our models optimized on different loss functions, rendered as contour plots. We overlay contour outlines of the real SDF, to show

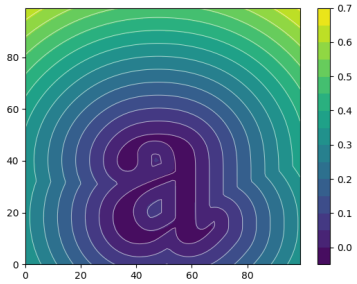
the difference between the predicted and real data. Aside from the MSE and our own loss function, we also include the loss function used by Park et al [14]. Looking closely at the thinnest parts of the shape, around $(50, 30)$ and $(50, 15)$, we can see gaps in the contour generated by the MSE-based model, which are not present with the model using our loss function. The model using the loss function from DeepSDF seems to produce a high quality 0-contour, but produces clearly visible errors at higher distances. As discussed, this could be a good trade-off depending on the use case, but our loss function seems to be able to avoid this trade-off.



(a) MSE



(b) Loss from DeepSDF



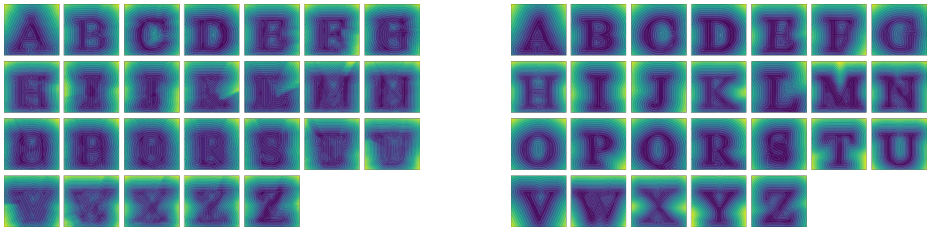
(c) Our loss function

Fig. 1: $SDF_{\texttt{a}}$ as learned by a neural network, with three different loss functions.

4.3 Multiple shapes

The next natural step is to learn multiple shapes with a single model, by training it on the glyphs from `A` through `Z`. We use a larger network, with four layers of 1024 nodes, to increase the capacity of the model for this more complex task. We train on $5 * 10^7$ points, using the same distribution as before. For each point

p_i in the training data we also choose a glyph s_i. The label is then chosen as $y_i = SDF_{s_i}(p_i)$. There are multiple representations that could be used for s_i. Figure 2 shows the result of two approaches. On the left, we encode each glyph with a single number, using $s_i = i$. On the right we assign a 32d vector to each glyph, uniformly sampled from $[-1, 1]^{32}$. We see that the latter yields much better results. With the single number encoding, glyphs tend to blend together with neighboring glyphs in the encoding space. Examples of this are `M` and `N`, and the range from `O` to `R`. With the higher-dimensional encoding, this problem seems to disappear, though some letters, such as `F` and `S`, are not clearly defined.



(a) SDF of multiple letters fitted by a neural network with an 1-dimensional glyph encoding

(b) SDF of multiple letters fitted by a neural network with a 32-dimensional glyph encoding

Fig. 2: SDF of multiple letters fitted by a neural network

4.4 Shape Embeddings

The random embeddings work surprisingly well, and enables our model to learn multiple shapes with few modifications. We think this success is in part because the model is able to learn to map certain features in the encoding vectors to features of the shape. Perhaps it learns that glyphs with open space in the center, such as `O`, `C` and `U` share a very small or large value for one of the 32 dimensions. This idea can be pursued further; instead of assigning static random vectors to each glyph, we can let the model update them during training. By backpropagating through the encoding vector, we allow the model to encode meaningful information. This technique was popularised in natural language processing, where word embeddings are trained on large corpuses, and have proven to be valuable for many different tasks. Because this technique is popular, it is also well supported in machine learning tool sets. TensorFlow provides a Keras `Embedding` layer that implements exactly this process, and can easily be added to our existing model.

To test the embeddings, we extend our test set to 62 shapes, including the glyphs for the lower and upper case alphabet, and the digits `0` through `9`. From the results, shown in Figure 3, the quality seems to have improved over the

random embeddings, even though there are more shapes to learn. There are still some artifacts near the edges, but all the shapes are clearly recognizable. This implies that the learned embeddings are effective in encoding a meaningful representation. We can make this clearer by projecting the embeddings into two dimensions using t-SNE. Figure 4 shows a t-SNE projection of our shape embeddings, shining some light on what the model has learned. Our first observation is that the 62 points are spaced fairly evenly, which shows that the model is able to clearly distinguish each shape. Looking closer at how the letters are distributed, we can see some patterns and clusters. In the bottom right, we see a group of letters with descenders (`q`, `g`, `p`, `y`), and in general, the right side of the plot contains almost all the lower case letters. Slightly above and to the right of the center, around (20, 40), we see a cluster of letters whose glyphs look close to a vertical line. There are many more patterns that emerge from this plot, but the final one we point out is how `0`, `C`, `O` and `G` are grouped near the center. That `o` is missing from this group implies that the learned features are not scale invariant.

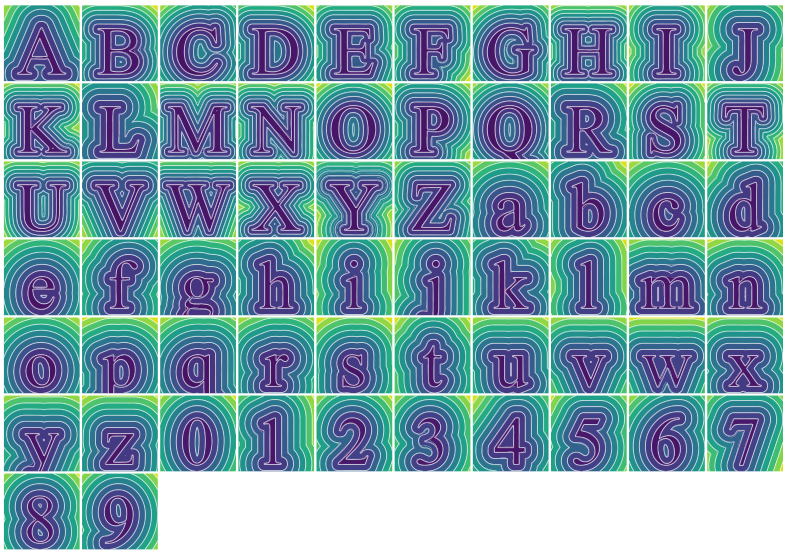


Fig. 3: Contour plots of model trained on 62 glyphs, with learned glyph embeddings.

4.5 Multiple classes

We extend our model once more, by allowing for classes with multiple shapes. For our data, we take the glyphs of our six fonts and use the character a glyph corresponds to as its class label. We extend our model input to take two embeddings: one for the glyph and one for the font. Figure 5 shows a few outtakes of

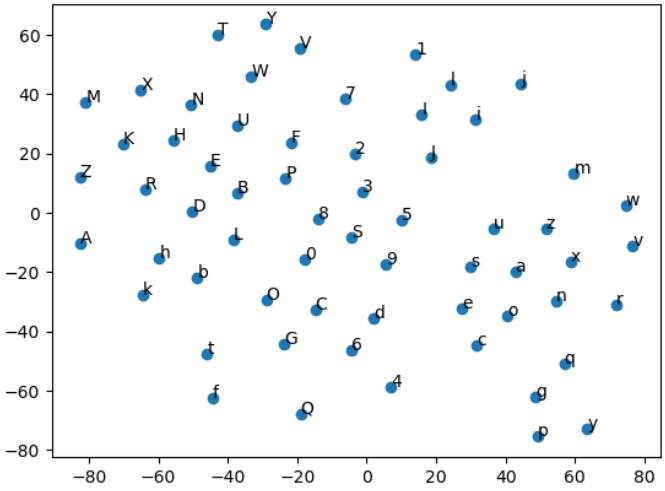


Fig. 4: t-SNE projection of the learned embeddings

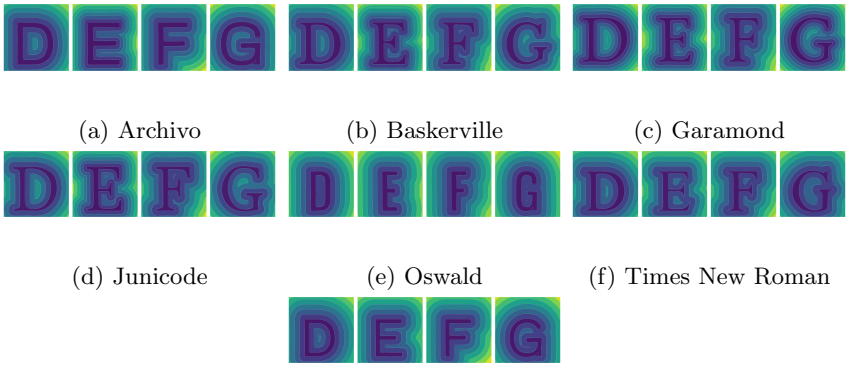
the results. We again see clear shapes and few artifacts. This shows the representational power and scalability of the embeddings. Because we train separate parameters for each glyph and each class, the model is able to scale to many characters and classes without modifications the model architecture.

4.6 Generating new shapes

The multi-class approach we used lays the ground for generating new shapes, as we can treat the two embeddings introduced in our previous model as an explorable latent space. By perturbing the font embedding, we can let the model generate alphabets for fonts that do not exist in the training data, while doing this with the glyph embedding results in new glyphs that should fit in the given font.

5 Exploring Latent Space

Our last model learns two 32d embeddings to represent glyphs and fonts. We can explore the learned embedding spaces and use them as a latent space to generate new, unseen shapes. We show two ways to choose new latent vectors: we can interpolate between known points in the space, and we can perturb a single dimension of a known point to see its effect. Because we have two embeddings, we can apply both techniques on the glyph embedding and the font embedding to get different results.



(a) Archivo (b) Baskerville (c) Garamond

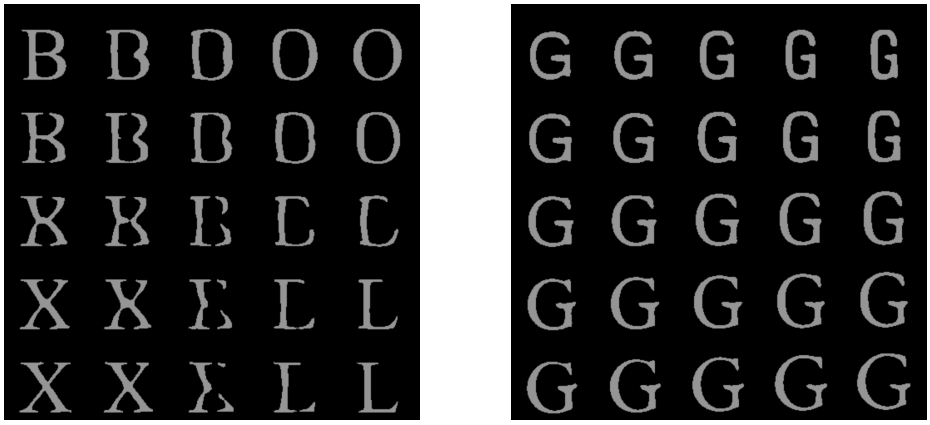
(d) Junicode (e) Oswald (f) Times New Roman

(g) Work Sans

Fig. 5: An SDF model trained on 26 classes with seven per class. We show `D` through `G`, in each of the seven fonts.

To show the effect of shape interpolation, we choose four latent vectors from our embeddings and create a 5×5 grid. For each grid cell we apply a bilinear interpolation between the four chosen vectors, to create a new latent vector. This vector is then applied to our model, yielding a new shape. The result is shown in Figure 6. To make the shape boundary more clear, we render the shapes in black when $SDF(p) > 0$ and white when $SDF(p) \leq 0$, instead of drawing a contour plot. The interpolation in glyph space is not as smooth as we had hoped. Though there is clear shape interpolation, most of the new shapes are not very convincing letters, and the ones that could pass as one do not look like a *new* letter. On the other hand, interpolating in font space works much better. This is not surprising, as the shape difference between a `G` in one font and another is smaller than the difference between a `X` and `L` in the same font.

To show the effects of manipulating our latent vectors in specific dimensions, we use a similar approach. We choose an embedding as our starting point in latent space, and choose two dimensions to manipulate. We draw a grid with the starting point in the center, increasing/decreasing one dimension for each column to the right/left while doing the same for the rows down and up. The result is shown in Figure 7. Similar to the interpolation results, the results for the glyph latent space are rough and do not yield convincing new letter shapes. The results for manipulating the font latent space are much more exciting. Many of the glyphs shown are convincing `G`s, and the two dimensions that we chose to manipulate seem to have a consistent interpretation: moving from top to bottom, we seem to be controlling the serif style, while moving from left to right we seem to be controlling the size of the glyph.



(a) B, O, X, and L in Garamond

(b) `G` in Work Sans, Oswald, Times New Roman and Garamond

Fig. 6: Interpolating in glyph space (left) and font space (right)

6 Future Research

SDFs are an exciting new tool for machine learning, and our work is only scratching the surface of what is possible. Future research could focus on finding a richer and smoother shape latent space, perhaps through some form of regularization on the embeddings. This is especially interesting in the context of PCG, since this would allow for generating richer shape content to use in games. Looking at how our method performs with different dataset could be interesting as well. For example: generating new emoji or different kinds of icons.

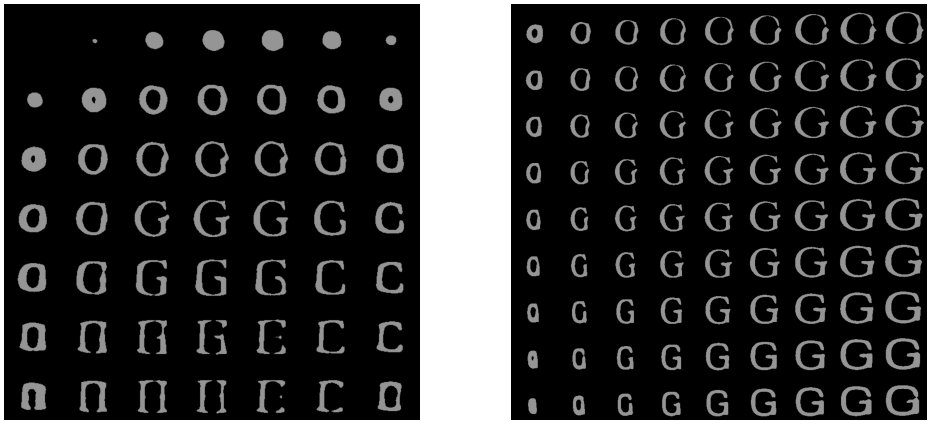
We propose a new loss function, that seems effective, but further research is needed to quantitatively compare our loss function with others.

Our work uses a simple fully connected neural network architecture. We expect more advanced architectures will be able to model SDFs more efficiently and with higher precision.

The embeddings itself are interesting in their own right. Much of the research and applications from word vectors should transfer to our shape embeddings. The font embeddings could be used for automatic classification or similarity search, and it might be possible to transfer characteristics from one font or glyph to another, by applying arithmetic on their embeddings.

It should be straight forward to translate our work to higher dimensional inputs, just by adding extra input nodes. Further research could look at the effectiveness of our custom loss function in representing objects in 3 dimensions or higher.

The way we train our model could also be improved. The shape produced by a SDF is defined by where the sign changes, by using this knowledge and the SDF itself it is probably possible to generate more efficient sets of points to



(a) `G` in Times New Roman, ±0.75 in dimensions 5 and 11 of the glyph latent space

(b) `G` in Times New Roman, ±0.25 in dimensions 11 and 30 of the font latent space

Fig. 7: Interpolating in glyph space (left) and font space (right)

train our network on than by just sampling from a normal distribution. Since different regions contain more or less amounts of information regarding to the shape.

7 Results & Conclusion

Signed Distance Functions provide a flexible and powerful implicit representation of shape geometry, while also being easily learned by simple fully connected neural networks. Our work explores their use in font learning and generation and determines weaknesses and strong points of using them in the domain. We found that we could effectively learn many fonts and glyphs with a single model, using learned embeddings. We were also able to convincingly generate new fonts or fonts that are a mixture between existing fonts. Generating new glyphs was much harder, and we were not able to consistently generate shapes that could pass as a new letter. We think this is largely due to sparseness in the available glyphs.

Our work can be extended directly to support other source geometry, such as bitmaps and vector images. We think our method could be the basis of a PCG system that can generate shapes with a lot of variation, while allowing content authors to condition the system to produce shapes with different classes of attributes.

8 Code

Our code can be found on GitHub [3].

References

1. Demoscene – the art of coding, http://demoscene-the-art-of-coding.net/
2. Open foundry, https://open-foundry.com
3. Sdfnet, https://github.com/mihaighidoveanu/sdf-game-net
4. et al, S.B.: Stb truetype, https://github.com/nothings/stb/blob/master/stb_truetype.h
5. Chen, Z., Zhang, H.: Learning implicit fields for generative shape modeling. CoRR **abs/1812.02822** (2018), http://arxiv.org/abs/1812.02822
6. Foo, M.J., Tiseo, C., Ang, W.T.: Application of signed distance function neural network in real-time feet tracking. In: 2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC). pp. 1191–1196 (2019)
7. Green, C.: Improved alpha-tested magnification for vector textures and special effects. ACM SIGGRAPH 2007 Papers - International Conference on Computer Graphics and Interactive Techniques (08 2007). https://doi.org/10.1145/1281500.1281665
8. Groueix, T., Fisher, M., Kim, V.G., Russell, B.C., Aubry, M.: Atlasnet: A papier-mâché approach to learning 3d surface generation. CoRR **abs/1802.05384** (2018), http://arxiv.org/abs/1802.05384
9. Guo, X., Li, W., Iorio, F.: Convolutional neural networks for steady flow approximation. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 481–490. KDD '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2939672.2939738, https://doi.org/10.1145/2939672.2939738
10. Hayashi, H., Abe, K., Uchida, S.: Glyphgan: Style-consistent font generation based on generative adversarial networks. CoRR **abs/1905.12502** (2019), http://arxiv.org/abs/1905.12502
11. Liu, S., Zhang, Y., Peng, S., Shi, B., Pollefeys, M., Cui, Z.: Dist: Rendering deep implicit signed distance function with differentiable sphere tracing (2019)
12. Mescheder, L.M., Oechsle, M., Niemeyer, M., Nowozin, S., Geiger, A.: Occupancy networks: Learning 3d reconstruction in function space. CoRR **abs/1812.03828** (2018), http://arxiv.org/abs/1812.03828
13. Michalkiewicz, M., Pontes, J.K., Jack, D., Baktashmotlagh, M., Eriksson, A.P.: Deep level sets: Implicit surface representations for 3d shape inference. CoRR **abs/1901.06802** (2019), http://arxiv.org/abs/1901.06802
14. Park, J.J., Florence, P., Straub, J., Newcombe, R.A., Lovegrove, S.: Deepsdf: Learning continuous signed distance functions for shape representation. CoRR **abs/1901.05103** (2019), http://arxiv.org/abs/1901.05103
15. Quilez, I.: 2d sdf functions, https://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm
16. Wu, Z., Song, S., Khosla, A., Tang, X., Xiao, J.: 3d shapenets for 2.5d object recognition and next-best-view prediction. CoRR **abs/1406.5670** (2014), http://arxiv.org/abs/1406.5670