Learning to Draw Vector Graphics: Applying Generative Modeling to Font Glyphs

by

Kimberli Zhong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Frédo Durand
Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chair, Master of Engineering Thesis Committee

Learning to Draw Vector Graphics: Applying Generative Modeling to Font Glyphs

by

Kimberli Zhong

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

TODO:

Thesis Supervisor: Frédo Durand
Title: Professor

Acknowledgments

TODO: acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TODO: hook

Our motivation for examining the *generation* of designs is two-fold. One, we see practical purpose in a recommendation tool that augments a designer's experience, and we believe such a tool would be a valuable addition to a designer's creative process. Two, we believe demystifying their generation would further solidify understanding of the intent and structure of human-created designs. In algorithmically mimicking the process by which glyphs are created, we hope to gain a deeper grasp of from where humans' notion of design and style arises.

Although many strides have been made in understanding and synthesizing rasterized images and designs, primarily with convolutional neural networks, we focus our investigation on the domain of vectorized images in this work. The two representations are quite different, and we aim to both produce generated designs with fewer aesthetically displeasing artifacts as well as investigate what new information about designs' underlying shapes and structures can be quantified and learned with the vectorized data format.

In the next chapter, we provide further background on the domain and discuss related work. Then, in the following chapters, we delve into the methods used to train our vector graphics generator on the data processing side as well as the model architecture side. We then demonstrate our methods as applied to font glyph generation, using a single-class as well as a multi-class approach. Finally, we evaluate our

results quantitatively and qualitatively and consider future directions of work.

Chapter 2

Background

The problem space we explore ties together work across a number of different disciplines. In hoping to apply generative methods to vectorized glyphs, we are inspired by previous work in computer vision on non-natural images and draw upon recent developments in generative modeling of line drawings.

2.1 Non-natural images

Natural images are photographs of real-world scenes and objects and to date have been the focus of research in many problems in computer vision, including object recognition, scene segmentation, and image generation. In contrast, non-natural images are computationally generated, either by hand with a computer design tool or automatically. Images in this category include graphs, pictograms, virtual scenes, graphic designs, and more. Algorithmically understanding, summarizing, and synthesizing these images pose unique challenges because of the images' clean and deliberately drawn designs, sometimes multimodal nature, and human-imbued intent.

Interest in applying computer vision methods to non-natural images is growing. Much progress has been made towards computationally understanding non-natural images on datasets including XML-encoded scenes [1], comic strips [2], and textbook diagrams [3]. Recent work by our group has explored the space of infographics, complex diagrams composed of visual and textual elements that deliver a message [4].

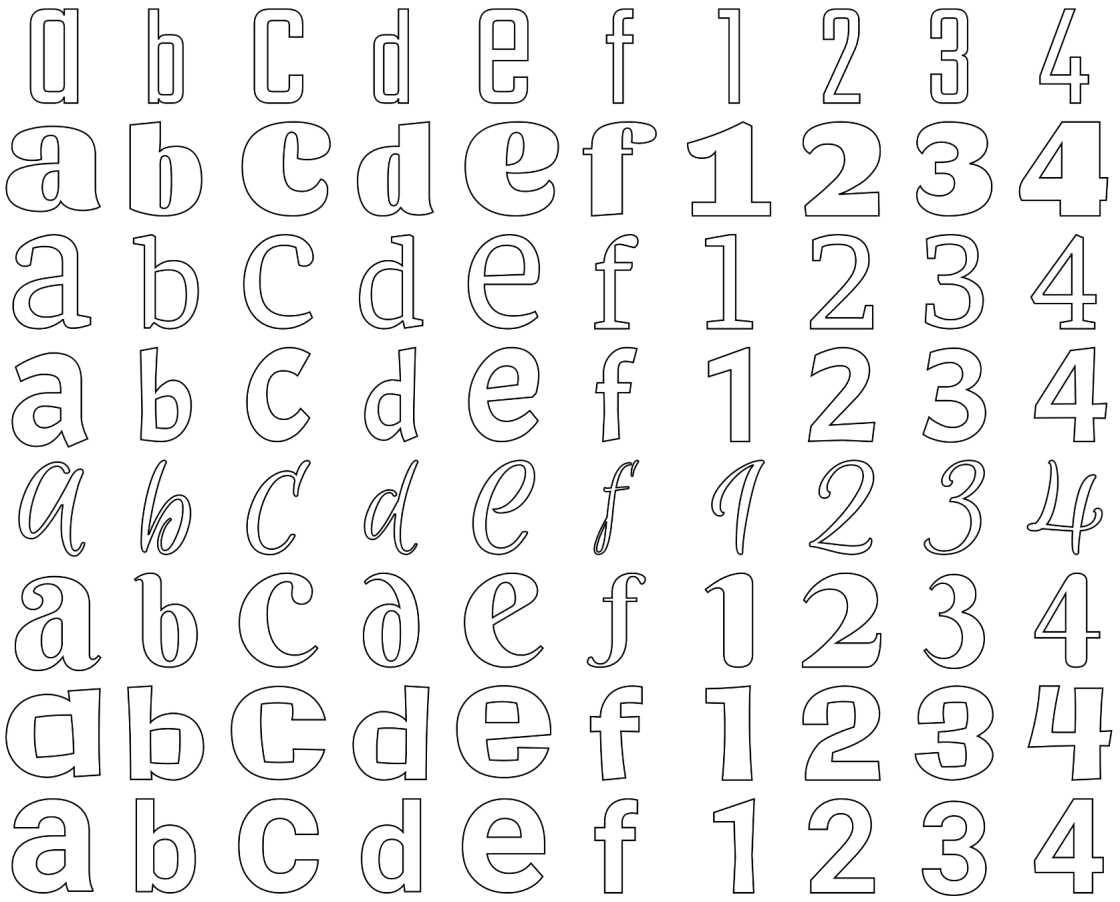


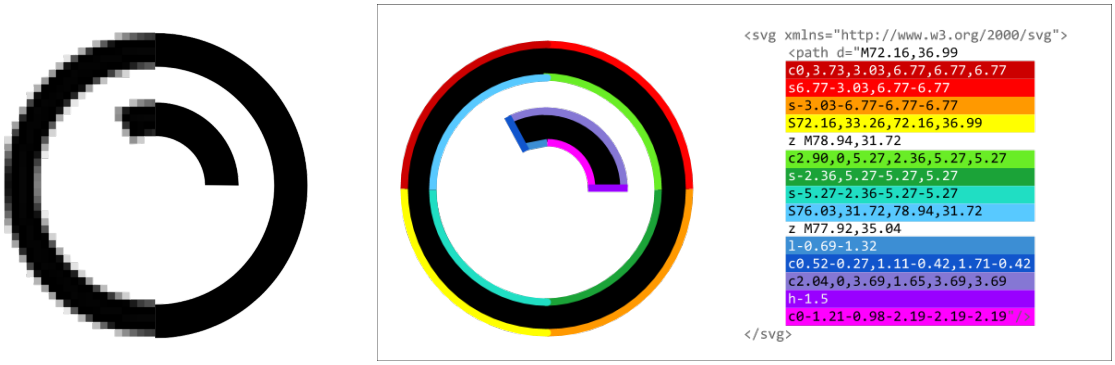
Figure 2-1: A sample of the types of font glyphs used in our dataset. Lowercase letters "a" through "f" are shown, as well as digits 1 through 4. Note the variety of styles represented: the dataset includes serif, sans serif, display, and handwriting style fonts.

In this work, we focus specifically on human-crafted font faces.

2.1.1 Font faces

Our computational methods are applied to font faces downloaded from Google Fonts TODO: cite. In Figure 2-1, a sampling of glyphs from the Google Fonts dataset is shown.

As a domain, font glyphs offer certain advantages and pose distinctive challenges. Unlike multimodal infographics and graphic designs, they are drawings essentially composed of simple lines and curves and, as such, can be modeled as sequential drawing commands. TODO: challenges



(a) Raster images are defined as two-dimensional arrays of pixel values, while vector graphics define curves and paths mathematically. Thus, when scaled, vector graphics (right) can still be rendered smoothly while raster images (left) degrade in quality.

(b) SVG is an XML-based markup language that describes geometric elements within a vector image. A single path is used to draw this soap bubble, and colored curves in the image correspond approximately to the highlighted attribute commands that describe them. For example, the command `l-0.69-1.32` indicates a line drawn from a starting point to a point 0.69 units to the left and 1.32 units down.

Figure 2-2: a visual comparison of raster and vector graphics. Figure 2-2b walks through a sample vector graphics path. Image source: *Dog Wash* by Llisole from the Noun Project.

2.1.2 Vector graphics

In contrast to raster images, which encode color values for each point (or *pixel*) in an image, vector graphics describe a parameterized set of curves and shapes.

Many specifications exist for describing images in vector format, including Scalable Vector Graphics (SVG), Encapsulated PostScript (EPS), and Adobe PDF.

2.2 Generative modeling

While discriminative modeling methods in machine learning focus on separating and identifying inputs to produce output labels learned from high-dimensional data such as images, generative algorithms create previously unseen instances of a class based on representative inputs. They are trained in an unsupervised or semi-supervised manner to model a data distribution P, estimating the true data distribution P_{gt} from which samples are drawn. By drawing probabilistically from P, they can then

be used to synthesize novel, unseen examples similar to input data.

Two popular neural network-based approaches include generative-adversarial networks (GANs) [5] and variational autoencoders (VAEs). Introduced in 2014 [6], GANs pit a generative model $G(z; \theta_g)$ against an adversary $D(x; \theta_g)$ that learns to discriminate samples from the ground truth dataset and the generative model's latent space. When both models are differentiable functions, backpropagation can be used to train G and D towards convergence in a computationally efficient manner.

2.2.1 Variational autoencoders

Variational autoencoders, introduced in [7], explicitly learn an encoder function mapping training examples from the input space $\mathcal{X}$ to a latent space $\mathcal{Z}$ as well as a decoder function mapping from $\mathcal{Z}$ to data points that are in $\mathcal{X}$ but distinct from training inputs X. TODO: explain more VAE math

In our work, we specifically build upon the variational autoencoder method in [8]. TODO: explain VAE + RNN

2.3 Related work

2.3.1 Image generation

- DRAW - SPIRAL - UToronto handwriting generation

2.3.2 Font style

- Learning a Manifold of Fonts

Chapter 3

SVG feature representation

Our goal is to extend beyond polyline modeling and capture the higher-level shapes of SVG objects. Thus, our first challenge is to choose an adequate representation that captures all drawing information from an SVG and can be fed as an input vector to our neural network architecture. Although many different elements are allowed by the SVG specification, we simplify the problem space to focus only on `path`s (since `path`s can be used to compose other elements anyway).

3.1 Overview of SVG commands

There are five commands possible in an SVG `path` [9]:

1. **Move**: moves the pen to a specified position (`m dx dy`)
2. **Line**: draws a line from the start to the end position (`l dx dy`)
3. **Cubic Bézier**: draws a cubic curve according to given control points (`c dx1 dy1, dx2 dy2, dx dy`)
4. **Quadratic Bézier**: draws a quadratic curve according to given control point (`q dx1 dy1, dx dy`)
5. **Arc**: draws a section of an ellipse (`a rx ry x-axis-rotation large-arc-flag sweep-flag dx dy`)

For simplicity, the above enumeration omits the absolute coordinate variants of the previous commands, which specify absolute (x, y) coordinates instead of relative (dx, dy).

3.2 Modeling SVGs

We would like to model SVG inputs without loss of information about pen movements. In essence, since SVGs name ordered lists of paths and their drawing commands, we model them as a sequence of mathematical parameters for the pen drawing commands and add a command for representing the transition between paths. The sequential nature of this representation makes the generation task well-suited to a recurrent neural network architecture, as we cover in Section 4.2.

3.2.1 Preprocessing

SVG icons and font glyphs often have additional properties like stroke and fill style. As we focus on path generation exclusively, our first step in transforming input SVGs is to strip away those styles and focus only on the `path` elements of the input, often resulting in an image that depicts the outlines of the input shape—see Figure 2-1 for examples.

Often, designers do not craft SVGs by editing XML by hand but rather with interactive image editing software such as Adobe Illustrator or Inkscape (TODO: citations?). Thus, human-created SVGs often have varying path compositions, viewboxes, and canvas sizes.

To constrain the generation process slightly, we first homogenize our dataset by preprocessing inputs to rescale to the same overall canvas size (set to 256×256 pixels) as well as reorder paths in the drawing sequence so that paths with larger bounding boxes are drawn first.

There is also variability in the direction of path drawing and in starting positions. Instead of controlling these factors in the preprocessing stage, our architecture is designed for bidirectionality, and all command sequences are prepended such that the

pen starts at coordinate $(0, 0)$.

3.2.2 Simplifying path commands

As seen in our dataset statistics in Section 4.1, the most common SVG path command used in our input data is the cubic Bézier drawing command, `c`, by a large margin. To avoid bias and to constrain the problem space (TODO: wording?), we instead consolidate the different path commands into a single feature representation.

Out of the five SVG path commands, three are parametic equations of differing degrees, so we can model these three (lines, quadratic Béziers, and cubic Béziers) using the parameter space for the highest degree cubic-order equation. An elliptical arc segment, on the other hand, cannot be perfectly transformed into a cubic Bézier. Arcs have five extra parameters used to describe them (x-radius, y-radius, angle of rotation, the large arc flag, and the sweep flag), but they occur relatively rarely in the dataset, so it makes sense to approximate them with the same parameter space as used for our parametric commands. We use the following method to approximate arc segments as cubic Béziers (further mathematical explanation can be found in Appendix A).

1. Extract the arc parameters: start coordinates, end coordinates, ellipse major and minor radii, rotation angle from x and y axes, large arc flag, and sweep flag.

2. Transform ellipse into unit circle by rotating, scaling, and translating, and save those transformation factors.

3. Find the arc angle from the transformed start point to end point on the unit circle.

4. If needed, split the arc into segments, each covering an angle less than 90°.

5. Approximate each segment angle's arc on a unit circle such that the distance from the circle center to the arc along the angle bisector of the arc is equal to the radius, defining cubic Bézier control points.

6. Invert the transformation above to convert arcs along the unit circle back to the elliptical arc, transforming the generated control points accordingly.

7. Use these transformed control points to parameterize the output cubic Bézier curve.

After all path drawing command types have been transformed to use the parameters needed for modeling cubic Bézier segments, we can represent each SVG command as a feature vector comprising those parameters and a three-dimensional one-hot pen state vector, similar to the feature representation used in [8].

In all, our feature representation models each drawing command as a nine-dimensional vector (in contrast to the five-dimensional feature vector for polyline drawings in [8]). Six dimensions are used to model three x, y coordinate parameters of cubic Béziers, and three dimensions are reserved for the pen up, pen down, and end drawing pen states. In the next section, we examine the details of this feature transformation process and how its tweaks affect modeling performance.

3.3 Feature representation variability

We find that alternative forms of this feature adaptation process have varying "learnability", as training variants of the transformed inputs with the same model architecture produces outputs of differing quality.

Some examples of variation are:

- **Absolute vs. relative coordinates**: are start, end, and control points represented in terms of their absolute position in the drawing or their relative displacement between each other?

- **Coordinate system orientation**: if relative, do we specify displacement vectors in terms of the normal $\{(1, 0), (0, 1)\}$ basis, or do we transform their values such that they represent deviations from continuing in the same direction as the previous point?

Table 3.1: The differences between the five feature representations used for the encoding efficacy experiment. Each feature encoding uses six dimensions for the cubic Bézier parameters (two for each point vector) and three for the pen state (pen up, pen down, end drawing). Note that `s` represents the start coordinates of the curve, `e` represents the end coordinates of the curve, `c1` represents the coordinates of the curve's first control point, and `c2` represents the coordinates of the curve's second control point. `disp(a, b)` indicates displacement between points `a` and `b`, and `rot(v)` indicates that vector `v` has been applied a transformation to rotate its coordinate axes to be in the TODO: direction of the previous vector in the encoding.

encoding	feature vector description
A	`disp(s, e), disp(s, c1), disp(s, c2), pen_state`
B	`disp(s, c1), disp(c1, c2), disp(c2, e), pen_state`
C	`disp(s, e), rot(disp(s, c1)), rot(disp(c2, e)), pen_state`
D	`e, rot(disp(s, c1)), rot(disp(c2, e)), pen_state`
E	`e, c1, c2, pen_state`

- **Pairwise coordinate choice**: if relative, between which points in the curves' coordinate parameters do we measure displacement?

3.3.1 Encoding experiment

To investigate the effects of choices for the above questions, we train five models each with a different feature encoding for input and output drawings.

All models are trained using the same architecture as described in Section 4.2, for TODO: count steps. To generate the differently encoded inputs, the same base dataset of SVGs for the glyph "b" in various font faces is transformed to produce a set of feature vectors for each representation in Table 3.1. The base dataset is then partitioned randomly into 1920 training examples, 240 validation examples, and 240 test examples. SVGs whose total number of path commands is greater than 250 are pruned. Graphs depicting loss during the training process can be found in Appendix B.

Results

For each of the models, the model iteration with the best validation loss is selected for evaluation. Qualitative results from the encoding experiment can be found in

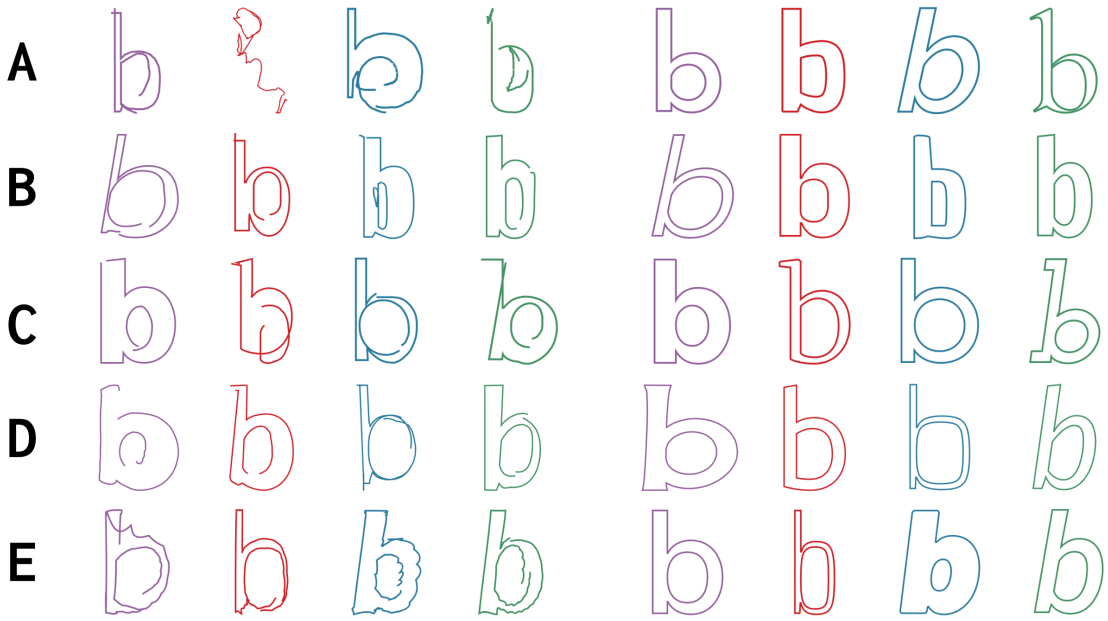


Figure 3-1: Randomly sampled input-output pairs from the SVG models trained on the five encodings described in Table 3.1. Decoded outputs were generated at temperature 0.3. Decoded outputs are found on the left, while ground truth inputs are on the right.

Figure 3-1.

We evaluate results quantitatively by TODO: eval!!!!.

Chapter 4

Font glyph generation

4.1 Dataset collection

4.2 Model architecture

4.2.1 Single-class

4.2.2 Multi-class

4.3 Training

4.4 Results

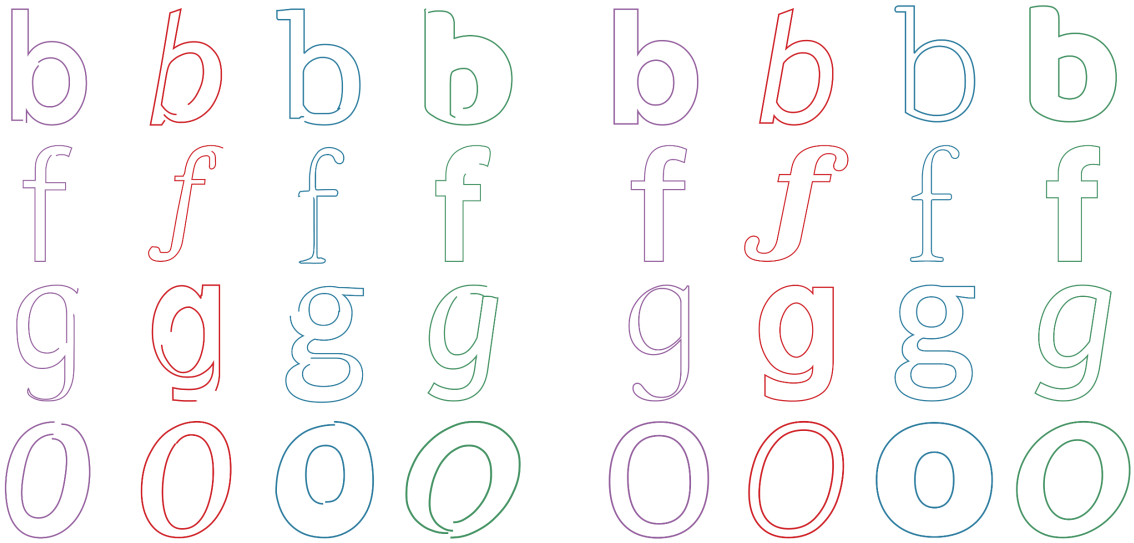


Figure 4-1: TODO: caption

Chapter 5

Evaluation

5.1 Qualitative results

5.1.1 Common modes of failure

- discussion of failures - vector interpolation

5.2 Quantitative results

Chapter 6

Conclusion

Appendix A

Algorithms

TODO: math for transforming arcs into cubics

Appendix B

Encoding experiment training

TODO: training graphs from encoding experiment

Bibliography

[1] J. Wu, J. B. Tenenbaum, and P. Kohli, "Neural scene de-rendering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 699–707.

[2] M. Iyyer, V. Manjunatha, A. Guha, Y. Vyas, J. Boyd-Graber, H. Daumé III, and L. Davis, "The amazing mysteries of the gutter: Drawing inferences between panels in comic book narratives," *ArXiv preprint arXiv:1611.05118*, 2016.

[3] M. J. Seo, H. Hajishirzi, A. Farhadi, and O. Etzioni, "Diagram understanding in geometry questions.," in *AAAI*, 2014, pp. 2831–2838.

[4] Z. Bylinskii, S. Alsheikh, S. Madan, A. Recasens, K. Zhong, H. Pfister, F. Durand, and A. Oliva, "Understanding infographics through textual and visual tag prediction," *ArXiv preprint arXiv:1709.09215*, 2017.

[5] A. Karpathy, P. Abbeel, G. Brockman, P. Chen, V. Cheung, R. Duan, I. Goodfellow, D. Kingma, J. Ho, R. Houthooft, T. Salimans, J. Schulman, I. Sutskever, and W. Zaremba. (2016). Generative models, [Online]. Available: `https://blog.openai.com/generative-models/`.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[7] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *ArXiv preprint arXiv:1312.6114*, 2013.

[8] D. Ha and D. Eck, "A neural representation of sketch drawings," *ArXiv preprint arXiv:1704.03477*, 2017.

[9] A. Grasso, C. Lilley, D. Jackson, J. Ferraiolo, P. Dengler, J. Watt, C. McCormack, J. Fujisawa, E. Dahlström, and D. Schepers, "Scalable vector graphics (SVG) 1.1 (second edition)," W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.