A CAD Interface for Drawing with Signed Distance Functions

Nicolas Schmidt
Input Experience

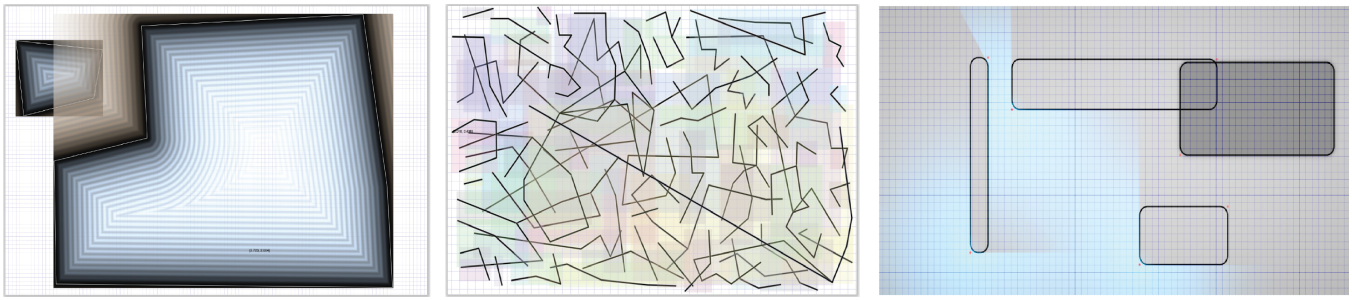

Figure 1: (a) Representation of polygon signed distance functions clipped to bounding plane (b) Many signed distance polylines on individual bounding planes (c) lighting effects with signed distance functions (Method 1)

ABSTRACT

This poster describes the implementation of a performant 2D drawing application in the browser that renders Signed Distance Functions (SDF) compiled from user input. SDFs are well suited for CAD applications because they reveal elegant boolean operations and efficiently allow for superior anti-aliasing. Because of their reliance on the GPU, SDFs have not traditionally lent themselves to graphical user input. By compiling shaders on the fly from user input we are able to seamlessly interact with rendered SDFs in a CAD interface.

CCS CONCEPTS

• **Applied computing** → **Computer-aided design**; • **Human-centered computing** → **User interface programming**.

KEYWORDS

WebGL, GUI, CAD, Signed Distance Function, Shader

ACM Reference Format:
Nicolas Schmidt. 2020. A CAD Interface for Drawing with Signed Distance Functions. In *SIGGRAPH '20 Posters: Special Interest Group on Computer Graphics and Interactive Techniques Conference Posters, August 24–28, 2020, Online.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3388770.3407435

1 INTRODUCTION

Signed Distance Functions (SDFs) are powerful expressions of geometry that parameterize forms by their distance from a point in space. These functions take a point as input and return the shortest distance to the surface of a shape. The sign of the distance will be positive outside the shape, 0 at the boundary of the shape, and negative inside the shape. In the 2D case of a circle this is a very simple function. The SDF for a circle centered on C is defined as $f(p) = dist(p, C) - radius$.

When this function is evaluated for many points p an image of the circle can be rendered, for instance, by coloring all points p where $f(p) == 0$ black. This produces a thin line in a circle around C. In the case of two overlapping circles, SDFs make it trivial to find the region representing their union. For two intersecting SDFs the union is represented by $min(f(p), g(p))$. In the case of two overlapping circles rendered as described previously, neither interior edge would be visible, instead the boundary where $p == 0$ would trace the union of the two intersecting regions.

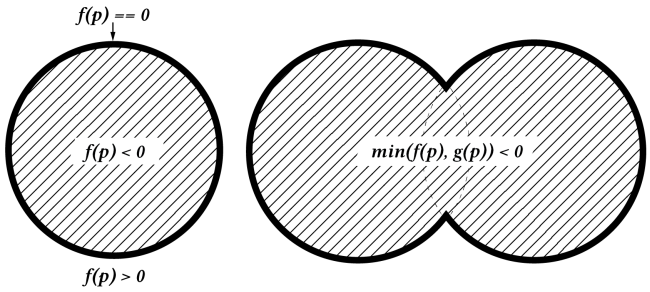

Figure 2: SDFs greatly simplify boolean operations such as the union opteration depicted above.

While SDFs have been used in graphics applications for decades, direct rendering of SDFs have been popularized through the demoscene, the work of Inigo Quilez and the website he co-created Shadertoy.com. One of the most fascinating things about SDFs is the simple form that typically complex geometric operations take when performed on SDFs. These include boolean operations such as union, intersection, subtraction [Frisken and Perry 2006], and other operations such as "annularization" and offsetting [Quilez [n.d.]]. Furthermore, SDFs provide very efficient and effective anti-aliasing

SIGGRAPH '20, August 24–28, 2020, Online

ACM ISBN 978-1-4503-7973-1/20/08.
https://doi.org/10.1145/3388770.3407435

[Green 2007]. These strengths should be of particular interest to CAD systems developers as these types of operations are central to parametric CAD workflows.

Previous experiments in interfaces for SDF authorship have relied on node-based or textual workflows [Lindborg et al. 2017] [Reiner et al. 2011]. Recently, SDFs have been used in 3D authorship for immersive art applications most notably *Dreams* for PS4 which uses a splatting approach that creates a painterly style [Evans 2015]. In the 2D space, SDFs have continued to be an area of research for acceleration of SVG rendering with work by Eric Lengyel showing how direct rendering of SDFs can lead to efficient rendering of pre-processed glyphs [Lengyel 2017].

2 APPROACH

The project's goals were to prototype a browser based drawing system that rendered SDFs from user input, took advantages of the composition operations SDFs enable, and benefitted from GPU acceleration. Two methods are described below, each of which achieves two of the project objectives.

2.1 Method 1

The first approach began by naively compiling a single fragment shader that took in a set of points stored in the RGB values of a texture as 16 bit half floats. The shader used to edit and draw the compiled scene were the same, and the program would re-compile this fragment shader when a primitive was baked to the scene.

The advantages of this approach were that users could get interactive visual feedback as they unioned primitives with the scene, changed parameters or added lighting simulations.

This approach led to the compilation of increasingly lengthy and complex fragment shaders. While rendering remained smooth for scenes with moderate numbers of primitives, shader compilation did not remain performant. As the shaders grew, the length of time to compile the shaders lengthened far too quickly and ultimately led this method to be abandoned. While this limitation led to its abandoment, *Method 1* allowed lighting simulations and SDF primitive booleans to be computed for many primitives in the same fragment shader at interactive framerates *(Figure 1c)*.

2.2 Method 2

The second attempt, which performs smoothly for large scenes uses a more traditional rendering pipeline in which each primitive shape drawn by the user is rendered on an individual bounding plane.

When a user begins drawing, a full screen quad is rendered using a primitive specific edit fragment shader. Similar to *Method 1* this shader takes a texture uniform that is populated by points from mouse clicks. Additionally, the current mouse position and parameters such as stroke weight, stroke color, fill color and opacity are passed to the shader via uniforms. When a user finishes drawing a primitive, the shape is baked to a new shader which is compiled from a primitive specific fragment stub. An axis aligned bounding box is computed for the primitive and the screen aligned quad is scaled and clipped accordingly *(Fig. 1a & 1b)*. The scene is represented by parallel arrays of points and primitives that are rendered to minimal quads and composited using WebGL blending.

While this method renders user input using SDFs very efficiently, because each primitve is drawn using a separate shader the interactive boolean operations and lighting effects enabled by *Method 1* are not easily realized.

3 CONTINUING WORK

In continuing work the benefits reaped from rendering SDF primitives by Method 1 and the efficiency realized by Method 2 must be reconciled. To create a scalable solution that does not fall prey to the same inefficiencies that made *Method 1* untenable, sampling of the SDF may be employed prior to composition. This could take the form of dynamically rendered 2D distance field textures as has been successful for text rendering [Green 2007]. Additionally, more primitive types including bezier or spline curves should be implemented to fully realize a robust tool-set for design authorship. Finally, tools should be built for primitive selection by mouse click, and box selection. Experimentation with CPU side evaluation of the SDF scene seems promising.

4 CONCLUSION

This paper presents an implementation of a CAD interface for authoring SDF scenes. Two methods are discussed, each with their own merits. Method 1 realizes the full benefits of SDF primitive composition and simulation techniques, even though shader compilation scales too quickly in relation to scene size. Method 2 successfully realizes a performant, graphical SDF drawing interface, but inhibits SDF primitive composition.

The strengths and limitations of *Method 1* and *Method 2* exemplify the difficulties of creating a graphical drawing interface for authorship using SDFs. The work presented in this paper represents a meaningful step forward in realizing such an interface.

ACKNOWLEDGMENTS

Thanks to Andrew Werner, Nikhilesh Sigatapu and Rob Sami for honest support.

REFERENCES

Alex Evans. 2015. Learning From Failure. www.youtube.com/watch?v=-3Yu0TCqa3Efeature=emb_title

Sarah F. Frisken and Ronald N. Perry. 2006. Designing with Distance Fields. In *ACM SIGGRAPH 2006 Courses* (Boston, Massachusetts) *(SIGGRAPH '06)*. Association for Computing Machinery, New York, NY, USA, 60–66. https://doi.org/10.1145/1185657.1185675

Chris Green. 2007. Improved Alpha-Tested Magnification for Vector Textures and Special Effects. In *ACM SIGGRAPH 2007 Courses* (San Diego, California) *(SIGGRAPH '07)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/1281500.1281665

Eric Lengyel. 2017. GPU-Centered Font Rendering Directly from Glyph Outlines. *Journal of Computer Graphics Techniques (JCGT)* 6, 2 (14 June 2017), 31–47. http://jcgt.org/published/0006/02/02/

Teemu Lindborg, Philip Gifford, and Oleg Fryazinov. 2017. Interactive Parameterised Heterogeneous 3D Modelling with Signed Distance Fields. In *ACM SIGGRAPH 2017 Posters* (Los Angeles, California) *(SIGGRAPH '17)*. Association for Computing Machinery, New York, NY, USA, Article 6, 2 pages. https://doi.org/10.1145/3102163.3102246

Inigo Quilez. [n.d.]. *2D distance functions.*

Tim Reiner, Gregor Mückl, and Carsten Dachsbacher. 2011. SMI 2011: Full Paper: Interactive Modeling of Implicit Surfaces Using a Direct Visualization Approach with Signed Distance Functions. *Comput. Graph.* 35, 3 (June 2011), 596–603. https://doi.org/10.1016/j.cag.2011.03.010