# A New Framework for Representing, Rendering, Editing, and Animating Type

Ronald N. Perry and Sarah F. Frisken, MERL


## Abstract

In this paper, we present a new framework for representing, rendering, editing, and animating character glyphs. Our framework is based on Adaptively Sampled Distance Fields (ADFs), which provide an ideal computational substrate for performing these operations. We introduce higher-order ADFs and describe why this succinct representation of 2-dimensional shapes strikes the optimal balance between memory use and computational load. We present a new antialiasing method, which exploits the distance field to achieve superior rendering quality for both static and animated type. We describe how perceptual rules used by professional type designers to create hand-tuned glyphs can be automatically applied during rendering in this framework, thus mitigating the labor-intensive process of manual hinting. For designing fonts within our framework, we provide an editor with a seamless interface between curve-based, stroke-based, and component-based design paradigms. Finally, we show how the distance field can be exploited to simulate interactions and behaviors that are difficult to compute directly from outline-based representations, thereby enabling creative design with animated type, some examples of which are provided.


**Keywords:** fonts, rasterization, antialiasing, hinting, grid fitting, distance fields.

## 1 Introduction

Today's industry standards for representing and rendering type are based on a vast history and legacy. These standards are well established and pervasive. However, two important trends in typography reveal some inherent limitations of current font representations, providing the impetus for change.

The first trend is the increasing emphasis of reading text on-screen due to the dominant role of computers in the office, the rise in popularity of Internet browsing at home, and the proliferation of PDAs and other hand-held electronic devices. These displays typically have a resolution of 72-100 dots per inch, which is significantly lower than the resolution of printing devices and unlikely to increase substantially in the near future. This low-resolution mandates special treatment when rasterizing type to ensure reading comfort and legibility, as evidenced by the resources that companies such as Microsoft and Bitstream have invested in their respective ClearType and Font Fusion technologies [Microsoft 2002; Thomas 2002].

The second trend is the use of *kinetic typography* (which requires animated type) in advertising, the web, and design [Lee et al. 2002; Small 1985; Wong 1995; Cho 1999]. Kinetic typography is used to convey emotion, to add interest, and to visually direct the viewer's attention. The importance of kinetic typography is demonstrated by its heavy use in television advertising and by the popularity on the web of Macromedia's Flash, which has over 400 million client seats [Macromedia 2003b].

Unfortunately, traditional outline-based fonts have limitations in both of these areas. Rendering type on a low-resolution display requires careful treatment in order to balance the needs of good contrast for legibility and reduced spatial and/or temporal aliasing for reading comfort. Outline-based fonts are typically *hinted* to

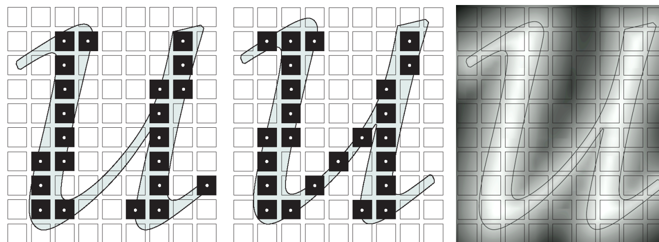

Figure 1. Pixel patterns generated by rasterizing the outline of an 18 point Palatino 'u' at 72 dpi. Left: a single sample per pixel exhibits jagged edges and dropout, both forms of aliasing. Center: a slight translation of the glyph results in a significantly different pixel pattern, the cause of temporal aliasing (i.e., flickering outlines and crawling jaggies) in animated type. Taking more samples per pixel reduces these artifacts but many samples (16+) are required for acceptable results. Right: the distance field of the 'u', where the signed distance from the outline is represented by image intensity (black is outside, white is inside). The distance field enables better antialiasing than its outline because sampled distances indicate the proximity of the outline even when they fall outside the shape. A single sample can provide a good estimate of how much of the glyph lies inside a filter footprint centered at each pixel. Furthermore, because the distance field is smooth, sampled distances change slowly as the glyph moves, reducing temporal aliasing.

provide instructions to the rendering engine for optimal rendering. Font hinting is labor intensive and expensive – developing a well-hinted typeface for Japanese or Chinese fonts (which can have 10,000+ character glyphs) can take years. In addition, because the focus of hinting is on improving the rendering quality of body type, these hints tend to be ineffective for type placed along arbitrary paths and for animated type. Furthermore, although high quality filtering can be used to antialias grayscale type in static documents that have a limited number of font sizes and typefaces, the use of filtering in animated type is typically limited by real-time rendering requirements [Ruehle and Halford 1999].

Here we introduce the use of Adaptively Sampled Distance Fields (ADFs) [Frisken et al. 2000] as a representation for type. The technical contributions of this paper include:

- A higher order interpolant that replaces the bi-linear interpolant of [Frisken et al. 2000; Perry and Frisken 2001] thereby reducing 2D ADF cell counts significantly and improving the distance field away from the edge of the glyph. The reduced cell counts allow the glyphs required for a typical Flash animation to fit in the on-chip cache of modern CPUs during processing; the improved field enables operations such as efficient collision detection for dynamic simulation of animated type.
- Methods for compressing ADFs that allow us to represent fonts with memory requirements comparable to hinted outline-based fonts and significantly smaller than bitmap fonts
- Two algorithms for rendering fonts with high quality spatial and temporal antialiasing that are faster and better than existing filtering methods, and which approach or surpass the quality of highly tuned glyphs rendered by Adobe Illustrator's proprietary font engine. The first algorithm uses a single sample per pixel while the second uses adaptive supersampling that is guided by the ADF data structure which provides a map of local variance in the distance field. Two implementations are described for each algorithm, a simple

scanline-based method and a cell-based method amenable to hardware.

- Two methods showing how ADFs can be used during rendering to automatically apply the perceptual hinting rules described in [Hersch et al. 1995] without requiring laborious manual hinting
- A system for designing fonts that allows type designers to move seamlessly between curve-based, stroke-based, and component-based design paradigms. The system includes an efficient new method for converting ADFs to Bezier curves as well as methods based on prior art for converting outlines and strokes to ADFs and for combining separately designed font components (such as stems and rounds) using constructive solid geometry (CSG). The system also provides an automatic method for generating ADFs from existing hand-drawn fonts and high-resolution digital masters.

Finally, we show how ADFs provide a computational substrate for simulating interactions between glyphs and for creating various effects that are difficult or inefficient to perform directly with outline-based representations. In particular, ADFs can be used to perform fast collision detection and impact force computation, soft-body deformation using implicit blends, and a wealth of effects available via level-set modeling (e.g., morphing, melting, smoothing, and fluid dynamics simulation).

## 2 Background

A typical Latin *font family*, such as Times New Roman or Arial, consists of a set of *fonts* (e.g., regular, italic, bold, and bold italic). Each font consists of a set of individual character shapes known as *glyphs*. There are a number of different ways to represent fonts, including bitmaps, outlines (e.g., Type 1 [Adobe Systems, Inc. 1990] and TrueType [Apple Computer, Inc. 1990]) and procedural fonts (e.g., Knuth's Metafont), with outlines being predominant. Hersch [1993] and Knuth [1979] contain comprehensive reviews of the history and science of fonts.

In this paper, we refer to two classes of typesize. *Body type* is type that is rendered at relatively small point sizes (e.g., 12 pt) and is used in the body of a document as in this paragraph. Body type requires high quality rendering for legibility and reading comfort. Size, typeface, and baseline orientation (e.g., horizontal) rarely change within a single document. *Display type* is type that is rendered at relatively large point sizes (e.g., 36 pt) and is used for titles, headlines, and in design and advertising to set a mood and to focus our attention. In contrast to body type, the emphasis in display type is on beauty (i.e., the lack of spatial and temporal aliasing) rather than legibility (where contrast may be more important than antialiasing). It is crucial that a framework for type be able to handle both of these classes well.

Type can be rendered to the display device as bi-level or grayscale. Some rendering engines use bi-level rendering for very small type sizes to achieve better contrast but perceptual studies have shown that well-hinted grayscale fonts are just as legible and are more highly rated by test subjects [O'Regan et al. 1996]. Hints are a set of rules or procedures stored with each glyph to specify how the glyph's outline should be modified during rendering to preserve features such as symmetry, stroke weight, and a uniform appearance across all the glyphs in a typeface [Hersch 1993]. Hinting is a time-consuming manual process; an excellent discussion of the complexities and difficulties of hinting can be found in [Zongker et al. 2000]. While there have been attempts to design automated and semi-automated hinting systems [Hersch 1987; Zongker et al. 2000], the hinting process remains a major bottleneck in the design of new fonts and in the tuning of existing fonts for low-resolution display devices. In addition, the complexity of interpreting hinting rules precludes the use of hardware for font rendering. The lack of hardware support forces compromises to be made during software rasterization (such as the use of fewer samples per pixel), particularly when animating type in real time [Ruehle and Halford 1999].

Grayscale font rendering typically involves some form of antialiasing, a process which smoothes out the jagged edges that appear in bi-level fonts. Although many font rendering engines are proprietary, most use supersampling (after grid fitting and hinting) with 4 or 16 samples per pixel followed by down-sampling via a 2x2 or 4x4 box filter respectively [Hersch et al. 1995; Ruehle and Halford 1999; Betrisey et al. 2000; Microsoft 2002]. This fairly rudimentary filtering is justified by the need for rendering speed. However, even this approach is too slow for real-time rendering (as required for animated type) and, as we will show, the rendered glyphs still suffer from spatial and temporal aliasing. In contrast, as will be demonstrated in Section 4, ADFs provide better antialiasing with only a single sample per pixel and very high quality antialiasing with only a few samples per pixel.

## 3 Representation

### 3.1 Background

For this paper, we define a 2-dimensional signed distance field $D$ representing a (closed) 2-dimensional shape $S$ (such as a glyph) as a mapping $D:\Re^2 \rightarrow \Re$ for all $\mathbf{p} \in \Re^2$ such that $D(\mathbf{p}) = \mathrm{sign}(\mathbf{p}) \cdot \min\{||\mathbf{p} - \mathbf{q}||$: for all points $\mathbf{q}$ on the zero-valued iso-surface (i.e., edge) of $S\}$, $\mathrm{sign}(\mathbf{p}) = \{-1$ if $\mathbf{p}$ is outside $S$, $+1$ if $\mathbf{p}$ is inside $S\}$, and $|| \cdot ||$ is the Euclidean norm. Less formally, the distance field of a glyph simply measures the minimum distance from any point **p** to the edge of the glyph, where the sign of this distance is negative if **p** is outside the glyph and positive if **p** is inside the glyph.

Figure 1 gives some intuition as to why the glyph's distance field provides better results than the glyph's outline for sample-based rendering. On the left, a single sample per pixel can miss the glyph even when the sample point is arbitrarily close to the outline. The rendered glyph has jagged edges and dropout, both forms of spatial aliasing. In the center, a slight translation of the glyph up and to the right results in a significant change in the sample values, the cause of temporal aliasing in animated type (i.e., flickering outlines and jagged edges that seem to crawl during motion). Taking more samples per pixel reduces these effects but many samples may be required for acceptable results [Mitchell 1996]. In contrast, on the right, sampled distance values indicate the proximity of the glyph even when they fall outside the shape. In fact, a single sample value can be used to estimate how much of the glyph lies inside a filter footprint centered at each pixel. Furthermore, because the distance field varies smoothly (i.e., it is $C^0$ continuous), sampled values change slowly as the glyph moves, reducing temporal aliasing artifacts.

Distance fields have other advantages. Because they are an implicit representation, they share the benefits of implicit functions [Bloomenthal 1997]. In particular, distance fields enable an intuitive interface for designing fonts: individual components of glyphs such as stems, bars, rounds, and serifs can be designed separately and then trivially blended together to compose different glyphs of the same typeface. Section 5 describes a system for editing glyphs. Distance fields also have much to offer in the area of kinetic typography since they provide information important for simulating interactions between objects. [Frisken and Perry 2002] summarizes the many contributions that distance fields have made to computer graphics and related fields. Some such uses of the distance field are described in Section 6.

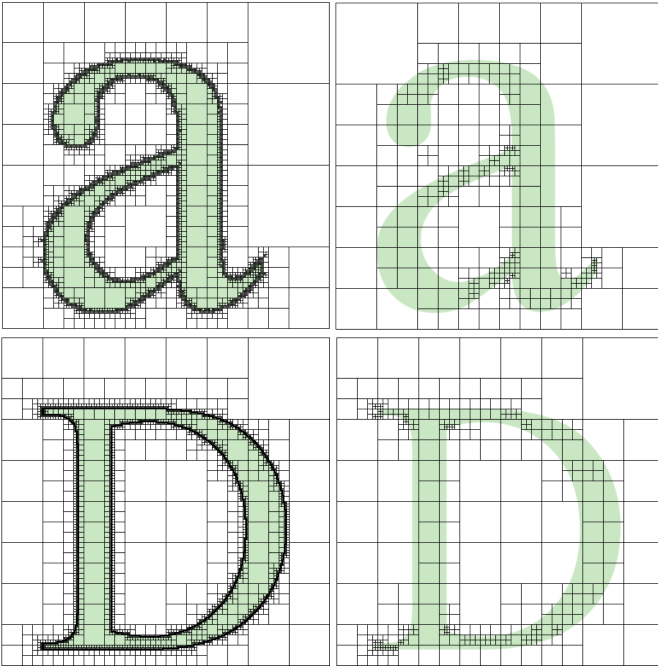

Figure 2. Times New Roman ‘a’ and ‘D’ represented as 3-color quadtrees (left) and bi-quadratic ADFs (right). The 3-color quadtrees for the ‘a’ and the ‘D’ consist of 17,393 and 20,813 cells respectively, while their corresponding ADF counterparts consist of 457 and 399 cells. Bi-quadratic ADFs typically require 5-20 times fewer cells than the bi-linear representation of [Frisken et al. 2000].

ADFs are an efficient digital representation of distance fields that use detail-directed sampling to reduce the number of samples required to represent the field, store the sampled distances in a spatial hierarchy for efficient processing, and provide a method for reconstructing the field from the sampled values. The use of detail-directed sampling (which samples the distance field according to its local variance) significantly reduces memory requirements over both regularly sampled distance fields (which sample at a uniform rate throughout the field) and 3-color quadtrees (which always sample at a maximum rate near edges).

The efficiency of ADFs is illustrated in Figure 2, which compares the size of a 3-color quadtree for a Times Roman ‘a’ and ‘D’ with the size of a new bi-quadratic ADF (presented in Section 3.2) of the same accuracy. Both quadtrees have a resolution equivalent to a 512x512 image of distance values.

### 3.2 Bi-Quadratic Reconstruction Method

Frisken et al. [2000] use a quadtree for the ADF spatial hierarchy and reconstruct distances and gradients inside each cell from the 4 distances sampled at the cell’s corners via bi-linear interpolation. They suggest that “higher order reconstruction methods … might be employed to further increase compression, but the numbers already suggest a point of diminishing return for the extra effort”. However, we have found that bi-linear ADFs are inadequate for the framework proposed in this paper. In particular, they require too much memory, are too inefficient to process, and the quality of the reconstructed field in non-edge cells is insufficient for operations such as dynamic simulation. The “bounded-surface” method of Perry and Frisken [2001] can be used to force further subdivision in non-edge cells by requiring that non-edge cells within a bounded distance from the surface (i.e., edge) pass an error predicate test. Although this reduces the error in the distance field within this bounded region, we have found that for bi-linear ADFs this method results in an unacceptable increase in the number of cells.

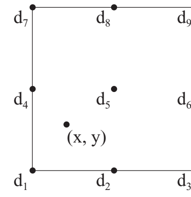


Figure 3. Each cell in a bi-quadratic ADF contains 9 distance values. The distance and gradient at (x,y) are reconstructed from these 9 values according to equations 1 – 3.

To address these limitations, we replace the bi-linear reconstruction method suggested in [Frisken et al. 2000] with a bi-quadratic reconstruction method. Bi-quadratic ADFs of typical glyphs tend to require 5-20 times fewer cells than bi-linear ADFs, where the higher reduction occurs when we require an accurate distance field in non-edge cells for operations such as dynamic simulation. This significant memory reduction allows the glyphs required for a typical Flash animation to fit in the on-chip cache of modern CPUs. This has a dramatic effect on times for processing glyphs because system memory access is essentially eliminated, easily compensating for the additional computation required by the higher order reconstruction method.

Figure 3 illustrates a bi-quadratic ADF cell. There are a variety of bi-quadratic reconstruction methods available: we use a bivariate Lagrange interpolating polynomial [Lancaster and Salkauskas 1990] which guarantees $C^0$ continuity along shared edges of neighboring cells of identical size. As with the bi-linear method, continuity of the distance field between neighboring cells of different size is maintained to a specified tolerance using an error predicate that controls cell subdivision during ADF generation [Perry and Frisken 2001]. The distance and gradient at a point (x, y), where x and y are expressed in cell coordinates (i.e., $(x, y) \in [0,1] \times [0,1]$), are computed as follows:

*Let* $xv_1 = x - 0.5$ *and* $xv_2 = x - 1$
*Let* $yv_1 = y - 0.5$ *and* $yv_2 = y - 1$
*Let* $bx_1 = 2xv_1 \cdot xv_2$, $bx_2 = -4x \cdot xv_2$, *and* $bx_3 = 2x \cdot xv_1$
*Let* $by_1 = 2yv_1 \cdot yv_2$, $by_2 = -4y \cdot yv_2$, *and* $by_3 = 2y \cdot yv_1$

$$\begin{aligned} dist = \; & by_1 \cdot (bx_1 \cdot d_1 + bx_2 \cdot d_2 + bx_3 \cdot d_3) + \\ & by_2 \cdot (bx_1 \cdot d_4 + bx_2 \cdot d_5 + bx_3 \cdot d_6) + \\ & by_3 \cdot (bx_1 \cdot d_7 + bx_2 \cdot d_8 + bx_3 \cdot d_9) \end{aligned} \tag{1}$$

$$\begin{aligned} grad_x = \; & -[by_1 \cdot (4x \cdot (d_1 - 2d_2 + d_3) - 3d_1 - d_3 + 4d_2) + \\ & by_2 \cdot (4x \cdot (d_4 - 2d_5 + d_6) - 3d_4 - d_6 + 4d_5) + \\ & by_3 \cdot (4x \cdot (d_7 - 2d_8 + d_9) - 3d_7 - d_9 + 4d_8)] \end{aligned} \tag{2}$$

$$\begin{aligned} grad_y = \; & -[(4y - 3) \cdot (bx_1 \cdot d_1 + bx_2 \cdot d_2 + bx_3 \cdot d_3) - \\ & (8y - 4) \cdot (bx_1 \cdot d_4 + bx_2 \cdot d_5 + bx_3 \cdot d_6) + \\ & (4y - 1) \cdot (bx_1 \cdot d_7 + bx_2 \cdot d_8 + bx_3 \cdot d_9)]. \end{aligned} \tag{3}$$

Reconstructing a distance requires ~35 floating-point operations (flops); reconstructing a gradient requires ~70 flops. Times reported in this paper do not exploit special CPU instructions such as the Streaming SIMD Extensions of Pentium class processors. However, because the reconstruction methods do not contain branches and the glyphs reside entirely in the on-chip cache, we can further optimize these reconstruction methods to take advantage of both special CPU instructions and the deep instruction pipelines of modern CPUs to make these methods extremely efficient.

### 3.3 Compression for Transmission and Storage

#### *3.3.1 Linear Quadtrees*

The spatial hierarchy (i.e., tree structure) of the ADF quadtree is

required for some processing (e.g., collision detection) but is unnecessary for others (e.g., cell-based rendering as described in Section 4.4). To provide better compression for transmission and storage of ADF glyphs, we use a linear quadtree structure [Samet 1989], storing the ADF as a list of leaf cells. The tree structure can be regenerated from these leaf cells at run-time as needed.

Each leaf cell in the linear ADF quadtree consists of the x and y positions (2 bytes each), the cell level (1 byte), the distance value at the cell center (2 bytes), and eight distance offsets from the center distance value (1 byte each), for a total of 15 bytes per cell. Each distance offset is computed by subtracting its corresponding sample distance value from the center distance value, scaling by the cell size to reduce quantization error, and truncating to 8 bits. The 2 bytes per position and the 1 byte for cell level can represent ADFs up to $2^{16}$ x $2^{16}$ in resolution, which are more than adequate for representing glyphs to be rendered at screen resolutions. Our tests have shown that such glyphs can be accurately represented by 16-bit distance values. Encoding eight of the distance values as 8-bit distance offsets provides substantial savings over storing each of these values in 2 bytes. Although, in theory, this may lead to some error in the distance field of large cells, we have not observed any visual degradation.

Typefaces require 500 – 1000 leaf cells per glyph for high-resolution ADFs. Applying lossless entropy encoding (e.g., Winzip) typically achieves 35-50% further compression. Consequently, an entire typeface of high-resolution ADFs can be represented in 300-500 Kbytes. If only body type is required or the target resolution is very coarse, as for cell phones, lower resolution ADFs can be used that require ¼ to ½ as many cells. These sizes are significantly smaller than grayscale bitmap fonts (which require ~0.5Mbytes per typeface for *each* point size) and are comparable in size to well-hinted outline-based fonts. Sizes for TrueType fonts range from 10's of Kbytes to 10's of Mbytes depending on the number of glyphs and the amount and method of hinting. Arial and Times New Roman, two well-hinted fonts from the Monotype Corporation, require 266 Kbytes and 316 Kbytes respectively.

### *3.3.2 Run-time Generation from Outlines*

ADFs can be generated quickly from existing outline descriptions (e.g., Bezier curves) using the tiled generator described in [Perry and Frisken 2001]. The minimum distance to a glyph's outline is computed efficiently using Bezier clipping [Sederberg and Nishita 1991]. Generation requires 0.04-0.08 seconds per glyph on a 2GHz Pentium IV processor and an entire typeface can be generated in about 4 seconds. Because conventional hints are not needed, the outlines required to generate ADFs are substantially smaller than their corresponding hinted counterparts. Therefore, rather than storing ADFs, we can store these minimal outlines and generate ADF glyphs from these outlines on demand. The reduced size of these minimal outlines is important for devices with limited memory and for applications such as Flash which transmit glyphs across a bandwidth-limited network.

### *3.3.3 Compression via Component-Based Fonts*

Bitstream achieves significant compression for Chinese, Japanese, and Korean fonts (which can consist of 10,000+ glyphs) by using a component-based representation in Font Fusion [Thomas 2002]. This representation decomposes glyphs into common strokes and *radicals* (i.e., complex shapes common to multiple glyphs), stores them in a font library, and then recombines them in the rendering engine. We note here that because distance fields are an implicit representation, ADFs can be easily combined using blending or CSG operations and thus are well suited to compression via this component-based approach.

# 4 Font Rendering

## 4.1 Background

### *4.1.1 Font Rendering*

In today's font rendering engines, fonts are predominantly represented as outlines, which are scaled on the fly to match the desired output size. While most high-resolution printers use bi-level rendering, modern display devices more commonly use grayscale rendering or a combination of grayscale and bi-level rendering at small point sizes (where the increased contrast of bi-level rendering is thought to improve legibility). A common approach for rasterizing grayscale glyphs involves scaling and hinting their outlines, scan converting the hinted outlines to a high-resolution image (typically 4 or 16 times larger than the desired resolution), and then down-sampling the high-resolution image by applying a filtering method (typically a box filter) to produce the final grayscale image [Hersch 1993].

For body type, individual glyphs can be rasterized once and stored in a cache as a grayscale bitmap for reuse. The need for sub-pixel placement of a glyph may require several versions of each glyph to be rasterized. Use of a cache for body type permits higher quality rendering especially since users will tolerate short delays (e.g., ½ second) during tasks such as paging through an Adobe Acrobat PDF document. However, type rendered on arbitrary paths and animated type are often rendered on the fly. Real-time rendering requirements force the use of lower resolution filtering (typically 4 samples per pixel and box filtering [Ruehle and Halford 1999]) which can cause static and temporal aliasing. To deal with complaints about the quality of their antialiased text, Macromedia's Flash provides a work-around for body type [Macromedia 2003a] that uses hinted *device fonts* (e.g., TrueType fonts residing on the client machine) instead of Flash's antialiasing method [Ruehle and Halford 1999]. Note, however, that Flash places severe constraints on how device fonts can be used in order to maintain real-time frame rates (e.g., device fonts cannot be scaled or rotated).

Recent work at Microsoft on ClearType has led to special treatment for LCD color displays that contain a repeating pattern of addressable colored sub-pixels. Platt [2000] presents a set of perceptually optimal filters for each color component. However, in practice, these optimal filters are implemented as a set of three displaced box filters (one for each color) [Betrisey et al. 2000]. We do not exploit this characteristic of LCD displays in this framework. However, the ADF antialiasing method described below could replace the box filters to provide better emulation of the optimal filters with fewer samples per pixel. This area of investigation is left as future work.

### *4.1.2 Antialiasing*

Understanding the artifacts in rendered fonts requires an understanding of antialiasing. Here we provide an overview from the unique perspective of font rendering and refer readers to some of the many excellent sources for a more in-depth treatment [Wolberg 1994; Akenine-Moller and Haines 2002; Joy et al. 1998]. Because pixels are discrete, rendering to a display device is inherently a sampling process, with the sampling rate imposed by the display resolution. Unless the sampling rate is at least twice the highest frequency in the source signal, the sampled signal will exhibit aliasing. Edges (e.g., glyph outlines) have infinite frequency components and hence cannot be represented exactly by sampled data. Inadequate sampling of edges results in *jaggies*, which tend to crawl along the sampled edges in moving images. If the source signal also contains a spatial pattern (e.g., the repeated

vertical stems of an ‘m’ or the single vertical stem of an ‘i’) whose frequency components are too high for the sampling rate, the sampled data can exhibit dropout as depicted in Figure 1, moiré patterns, and temporal flicker.

To avoid aliasing, the input signal must be pre-filtered to remove frequency components above those permitted by the sampling rate. In general, there are two approaches to pre-filtering. The first is known as analytic filtering. It applies some form of spatial averaging to a continuous representation of the source signal before sampling. Unfortunately, analytic filtering is often not possible, either because the source data is not provided as a continuous signal (the normal case for image processing) or because determining an analytic description of the signal within the filter footprint is too complex (the case for all but simple geometric shapes in computer graphics and certainly the case for spline-based outlines). The second approach is known as discrete filtering. In this approach, the source signal is sampled at a (typically) higher rate than the target rate and then a discrete filter is applied to reduce high frequencies in this supersampled image before down-sampling it to the target rate.

This discrete approach is referred to as regular supersampling in computer graphics. Various discrete filters can be applied depending on the processing budget, hardware considerations, and personal preferences for contrast vs. smoothness in the output image. The box filter typically used to render type simply replaces a rectangular array of supersampled values with their arithmetic average and is generally regarded as inferior in the signal processing community. Whitted [1980] introduced the use of adaptive supersampling, which focuses available resources for sampling and filtering on areas of the image with higher local frequency components. Lee et al. [1985] showed that the optimal adaptive sampling could be determined from the local variability in the image. However, Joy et al. [1998] note that “the usefulness of this technique is limited by the need to estimate the local variance of the image”. Cook [1986] argued that moiré patterns, which are introduced by inadequate regular sampling of high frequency patterns, are particularly objectionable to the human visual system. He proposed instead the use of stochastic or jittered sampling in which samples are randomly displaced slightly from their nominal positions. Jittered sampling tends to replace moiré pattern aliasing with high frequency noise and has been shown to be particularly effective in reducing temporal aliasing. As far as we know, neither adaptive supersampling nor jittered sampling have been applied to font rendering.

## 4.2 Rendering with Distance-Based Antialiasing

*4.2.1 Background*

The infinite frequency components introduced by the glyph edges are a major contribution to aliasing in font rendering. In contrast, a glyph’s 2D distance field does not contain such edges (see Figure 1). Instead, its maximum frequency depends on the spatial pattern of the glyph itself (e.g., the repeated vertical stems of an ‘m’ or the single vertical stem of an ‘i’). By representing a glyph by its 2D distance field we are effectively applying an analytic pre-filter to the glyph, although the distance field is different from the output of a conventional analytic pre-filter (which provides a filtering of coverage, i.e., a measure of how much of the filter footprint is covered by the glyph).

The use of distance to provide analytic pre-filtering for antialiasing is not new to computer graphics. Gupta and Sproull [1981] map distance to intensity in order to antialias thick lines in a Bresenham-like line drawing algorithm. The mapping relates distance to coverage by computing the convolution of a thick line with a cone filter as a function of distance from the centerline of the thick line to the center of the filter. Turkowski [1982] also maps perpendicular point-line distance to intensity. He applies his method to antialias lines and polygon edges with an efficient algorithm for computing distances to line segments. More recently, Jones and Perry [2000], building upon [Max 1990], used distances measured along line samples to estimate 2D coverage for polygons in 3D rendering. They note that the ideal line sample is perpendicular to a polygon’s edge and compensate for non-ideal line orientation by using multiple line samples. McNamara et al. [2000] describe a hardware implementation for antialiasing lines which maps distances to image intensity. Finally, distance filters (and related cone filters) have been used in volume rendering to “voxelize” volume models. Sramek and Kaufman [1999] present an analysis for why distance filters provide ideal pre-filtering for antialiased volume rendering.

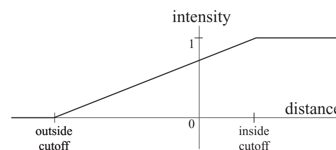

Figure 4. Profile of a linear filter used for converting ADF distances to image intensity. Distances are positive inside the shape and negative outside the shape. Different cutoff values (measured in pixels) affect the edge contrast and stroke weight.

*4.2.2 Antialiasing with ADFs*

Given an ADF of a glyph, it is both simple and efficient to apply distance-based antialiasing during font rendering. The ADF is rasterized into an image in four steps. For each pixel in the image, 1) the pixel coordinates are transformed into ADF coordinates; 2) the ADF cell containing the transformed pixel is located using the comparison-free quadtree traversal methods of [Frisken and Perry 2003]; 3) the distance at the transformed pixel is reconstructed from the cell’s distance values; and 4) the reconstructed distance is mapped to pixel intensity.

The mapping loosely relates the distance value to coverage – if we assume that the glyph’s outline is a straight line near the pixel, the mappings from distance to intensity used by Gupta and others hold. However, because we cannot make this assumption in general, we have experimented with several different mapping functions with the characteristics described in [Jones and Perry 2000; McNamara et al. 2000], including linear, Gaussian, and sigmoidal functions. Choice of the best mapping function is subjective but we prefer the linear mapping illustrated in Figure 4 followed by a contrast enhancement. The results shown here and on the accompanying webpages use a linear filter with outside and inside filter cutoff values of (-0.75, 0.75) pixels for display type and (-0.5, 0.625) pixels for body type. The contrast enhancement takes the form of a power function intensity_out = intensity_in$^{\alpha}$, with $\alpha$ set to 1.1. To accommodate a range of contrast preferences, $\alpha$ can be tuned by the user as is demonstrated in the Kanji examples provided on the webpages.

We have tested this antialiasing method on many different typefaces at many different point sizes. Some typical results are shown in Figures 5–7. Additional results for Latin and Kanji type rendered in font sizes ranging from 8 to 72 points are shown on the accompanying webpages. (Note: we urge reviewers to view the results from the webpages on a display device rather than from this PDF or its printed form.)

We have compared our method to other filtering methods, including 4x supersampling with box filtering, 16x supersampling with a more sophisticated (and computationally expensive) Gaussian filter, and 16x supersampling with the Mitchell-Netravali filter [Mitchell and Netravali 1988]. Because the Mitchell-Netravali filter provides only marginal improvement

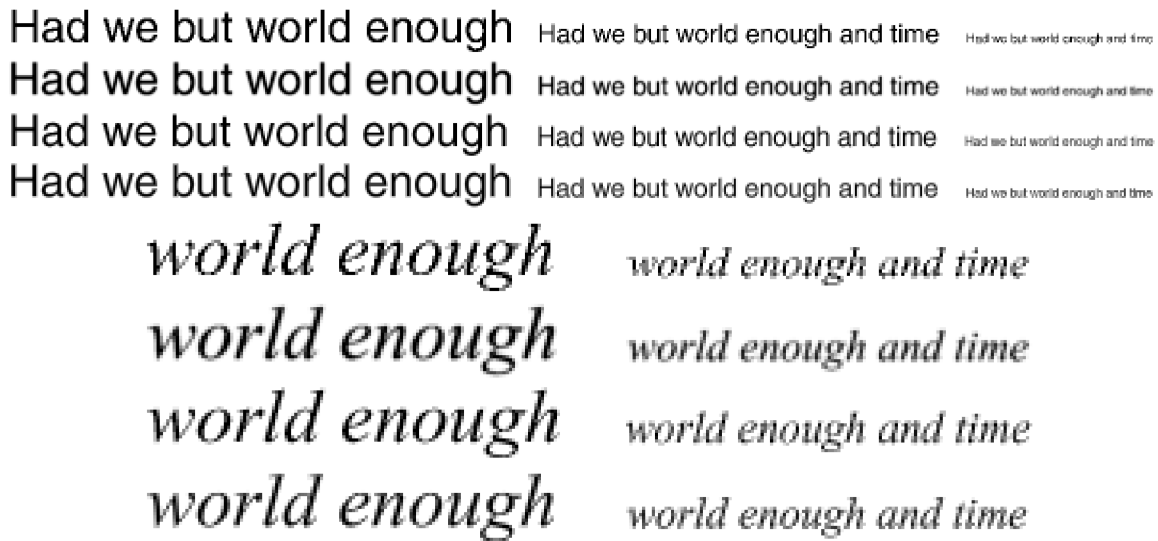


Figure 5. Examples of body type rendering for different typefaces and sizes. In each example 4 rows are rendered, from top to bottom, by 4x supersampling with a box filter, 16x supersampling with a Gaussian filter, Adobe Illustrator with hinting and supersampling, and the unhinted single sample per pixel ADF method of Section 4.2.2. Top: Helvetica typeface rendered at 3 sizes, captured from the display screen and pixel replicated 3 times. Bottom: Times New Roman italic typeface at 2 sizes, pixel replicated 5 times. In each example, note that 4x supersampling with box filtering produces blocky type with jagged edges and non-uniform stroke weights. The 16x Gaussian filtered type has fewer aliasing artifacts but the text is fuzzy and heavy compared to the hinted Adobe Illustrator type. In contrast, the ADF results are comparable in quality to the hinted Adobe Illustrator results.

Figure 6. Garamond glyphs at display type sizes rendered, from top to bottom, by 16x supersampling with a Gaussian filter, Adobe Illustrator with hinting and supersampling, and the unhinted single sample per pixel ADF method of Section 4.2.2. These images were digitally captured from the display device and then pixel replicated 3x. Note the ragged left edges of the vertical stems of the H, P, and J, the ragged horizontal bar of the H, and the ragged top bars of the P and E for the supersampled and Adobe Illustrator renderings. The ADF method produces smoother edges while maintaining good contrast.

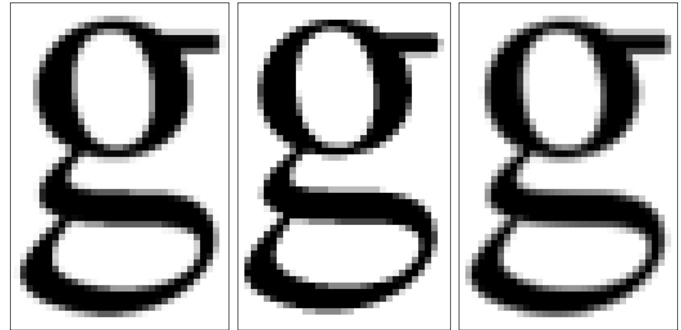


Figure 7. Times New Roman ‘g’ rendered, from left to right, by 16x supersampling with a Gaussian filter, Adobe Illustrator with hinting and supersampling, and the unhinted single sample per pixel ADF method of Section 4.2.2. These images have been pixel replicated 8x. Note the smooth grayscale transitions in the ADF rendering along the horizontal sections of the descender, resulting in better spatial and temporal antialiasing.

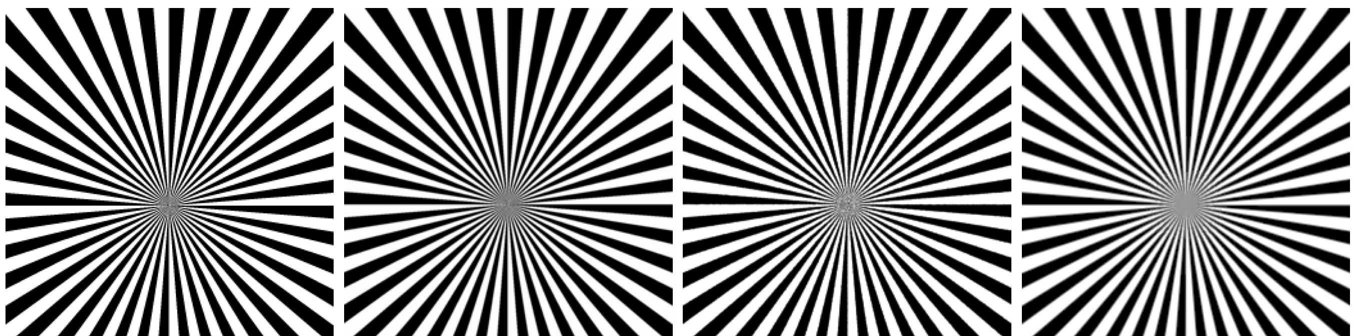

Figure 9. Test pattern rendered from left to right by 4x supersampling with a box filter, 16x supersampling with a Gaussian filter, 16x jittered supersampling with a Gaussian filter, and optimal adaptive ADF supersampling. The filtering methods of the two left images are used most often for antialiasing type. Adaptive ADF supersampling significantly reduces aliasing artifacts using an average of only 3.6 samples per pixel for this image. The dramatic reduction in temporal aliasing is best seen on the accompanying webpages.

over the Gaussian filter at a significant computational cost, these results are not shown. We have also compared our results to hinted, antialiased Adobe Illustrator fonts rendered with Adobe's highly tuned proprietary font rendering engine. We chose Adobe Illustrator as a gold standard over several other applications (including Microsoft Word, Adobe PhotoShop, and Adobe Acrobat) because the quality of Illustrator's font rendering was noticeably superior.

Figure 5 compares these rendering methods for body type sizes. Figures 6–7 present results for display type sizes. For glyphs, our method provides higher quality with a single sample per pixel and substantially shorter rendering times (4x–16x faster) than both supersampling methods. Our unhinted results are comparable to Adobe's proprietary rendering for well-hinted fonts (e.g., Arial and Times New Roman) for body type sizes and arguably better than their rendering at display type sizes.

### 4.3 Optimal Adaptive Supersampling

We have demonstrated that using the distance field reduces aliasing due to glyph edges. However, aliasing still occurs when stem widths or spacing between glyph components are too small for the display's sampling rate. In such cases, we can apply adaptive supersampling to further reduce spatial and temporal aliasing. Once again, ADFs have a significant advantage over outline-based representations – because ADFs use detail-directed sampling, regions of the distance field with higher local variance are represented by smaller leaf cells. Hence, the structure of the ADF quadtree provides the map of local variance required to implement Lee's optimal adaptive sampling, overcoming the difficulty noted by Joy et al. (see Section 4.1.2).

For each pixel in the image, the cell containing the pixel is located (as described in Section 4.2.2) and a set of sampled distances within a filter radius, r, of the pixel is reconstructed. The number of samples per pixel (spp) depends on the relative size of the cell, cellSize, to r. These sampled distances are filtered to produce a single weighted average for the pixel that is then mapped to pixel intensity. Various filters and sampling strategies are possible. Results presented in this paper were computed using a general form of the Gaussian [Wolberg 1994], weighting each distance sample by $W^{-1}2^{-3(d/r)^2}$, where d is the distance from the sample point to the pixel and W is the sum of the weights used for that pixel. We have achieved similar results using box filters, cone filters, and other forms of the Gaussian. The sampling strategy that we used is illustrated in Figure 8. Samples are placed in concentric circles about the pixel center for efficient computation of the weights and weight sums. We use a filter radius of 1.3 times the inter-pixel spacing and sample with 1 spp when cellSize > r, 5 spp when r/2 < cellSize ≤ r, and 13 spp when cellSize ≤ r/2. The adaptive method is not very sensitive to the sampling strategy: Section 4.4 describes another adaptive sampling strategy with equally good results that places sample points at the centers of all the cells contained within the filter radius.

Figure 9 compares screen shots of a demanding test pattern that has sharp edges and a spatial pattern at the image center with infinite spatial frequencies. The test pattern was rendered using adaptive ADF supersampling, 4x supersampling with a box filter, 16x supersampling with a Gaussian filter, and 16x jittered supersampling with a Gaussian filter. With adaptive ADF supersampling the edges are less jagged and the moiré pattern at the center of the image is significantly reduced. The dramatic improvement in both spatial and temporal aliasing is best demonstrated by an animation of the test pattern provided on the accompanying webpages. Display type also exhibits a significant

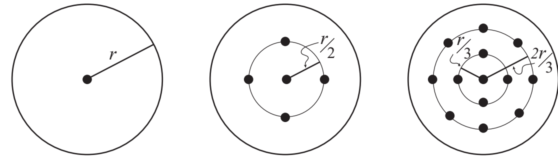


Figure 8. Sampling patterns used for adaptive supersampling. For each pixel, the cell containing the pixel is located and its size, cellSize, is determined. If cellSize > r (where r is the pixel filter radius) one sample is taken (left). If r/2 < cellSize ≤ r, 5 samples are taken (center). If cellSize ≤ r/2, 13 samples are taken (right).

reduction in temporal aliasing when rendered with adaptive ADF supersampling. The number of samples required for adaptive ADF supersampling depends on the spatial structure in the source signal and the target image size on the display device. The test pattern (an extreme case) requires an average of 1.2 spp @ 515x512, 3.6 spp @ 128x128, and 13 spp for very small images (8x8). Animated display type requires between 1 and 2.5 samples per pixel, significantly fewer than the 4–16 samples per pixel required by the other three methods.

### 4.4 Cell-Based Rendering

The ADF rendering algorithms presented in Sections 4.2 and 4.3 are easily implemented in software using scanline-based rasterization. Alternatively, ADFs can be rendered cell-by-cell (i.e., in object order). Cell-based rendering eliminates tree traversal for locating cells containing the sample points, eliminates redundant setup for computing distances and gradients within a single cell, and reduces repeated retrieval (i.e., memory fetches) of cell data. In addition, because the cells required for rendering can be represented as a sequential block of fixed sized (Section 3.3.1), self-contained (i.e., distances and gradients for points within a cell are computed from the cell's 9 distance values) units, a cell-based approach is amenable to hardware.

For the single sample per pixel ADF rendering method of Section 4.2, each leaf cell is rasterized. During rasterization distances are reconstructed at each pixel that falls within the cell and then mapped to intensity as described in Section 4.2. This approach is augmented with a special treatment of cells smaller than the filter radius for adaptive ADF supersampling. Because small cells occur where there is high variance in the distance field, distances in pixels near these cells should be pre-filtered before mapping them to intensity. In the spirit of object-order splatting [Westover 1990], we maintain a temporary buffer, initialized to zero, for accumulating the weights and the weighted distances for each pixel. As each cell is processed, the weighted distances and accumulated weights are incremented for each pixel that either lies in the cell or lies within the filter radius of the cell's center. After processing all the cells, the weighted distances are normalized by the accumulated weight for each pixel and then mapped to pixel intensity. We use the same Gaussian weights and filter radius as described in Section 4.3. Although this sampling method acquires different samples than the method of Section 4.3, the quality of the resulting antialiased images is similar.

As a final note, cell-based rendering always processes every leaf cell, regardless of the relative sizes of each cell to the filter radius. In theory, this provides optimal adaptive supersampling. In practice, the ADF quadtree can be used as a mipmap [Akenine-Moller and Haines 2002] to reduce the number of cells processed for small images. The ADF quadtree structure allows us to replace small leaf cells with their ancestors, effectively truncating the quadtree at some cell size. Our tests show that as long as this cell size is less than or equal to ¼ of the inter-pixel spacing, there is no visual degradation in the adaptive supersampling results.

### 4.5 Automatic Hinting

Hinting in standard font representations is a time-consuming manual process in which a type designer (and/or a hinting specialist) creates a set of instructions for better fitting individual glyphs to the pixel grid. Good hinting produces glyphs at small type sizes that are well spaced, have good contrast, and are uniform in appearance. Hersch et al. [1995] demonstrate that filtering methods produce fuzzy characters and assign different contrast profiles to different character parts, thus violating important rules of type design. According to Hersch et al., these rules include 1) the need to provide vertical stems with the same contrast distribution, with the left edge having the sharpest possible contrast, 2) the need for diagonal bars and thin, rounded parts of glyphs to have sufficient contrast for transmitting visual structure to the eye, and 3) the need for serifs to hold together and provide enough emphasis to be captured by the human eye. For outline-based fonts, rendering with hints is a three step process: 1) scaling the glyph's outlines and aligning them to the pixel grid; 2) modifying these outlines to control contrast of stems, bars, and serifs and to increase the thickness of very thin sections and arcs; and 3) supersampling the modified outlines followed by down-sampling with filtering.

Although the unhinted ADF rendering methods described above compare favorably with existing font rendering methods, research indicates that perceptual hinting can improve reading comfort at small type sizes [O'Regan et al. 1996]. Here we report some preliminary results that show that the ADF distance field can be exploited to provide automatic hinting.

The first step in hinting is to scale and align the glyph to the pixel grid. This can be done automatically from the given (or derived [Herz and Hersch 1994]) font metrics, i.e., the cap-height, the x-height, and the position of the baseline. Figure 10 illustrates the benefits of this simple process on an ADF glyph rendered at a small point size. After applying this simple form of grid fitting we can also use the distance field to provide some of the perceptual hints advocated by Hersch. Examples of two results are shown in Figures 11 and 12. In Figure 11, the direction of the gradient of the distance field is used to detect pixels on the left or bottom edge of a glyph. By darkening these pixels and lightening pixels on opposite edges, we achieve higher contrast on left and bottom edges without changing the apparent stroke weight. In Figure 12, we note that for pixels located on or near thin regions of the glyph, neighbors on either side of the pixel will have opposite gradient directions (i.e., their dot products will be negative). By detecting abrupt changes in gradient directions, we can darken pixels on these thin regions, providing better contrast for diagonal stems and thin arcs.

These are only two examples of how the distance field can be used to provide perceptual hints automatically without manual hinting. A full treatment of hinting (including using the distance field to provide optimal character spacing and uniform stroke weight) is beyond the scope of this paper and is left as future work.

### 5 Creating and Editing Fonts

There are two basic methods for designing fonts. The first is manual: glyphs are drawn by hand, digitized, and then outlines are fit to the digitized bitmaps. The second is by computer, for which there are three types of tools available: 1) direct visual tools for curve manipulation, 2) programmable design tools (such as Metafont) where the shape of a glyph is constructed by executing the instructions of a procedure which either a) defines a shape's

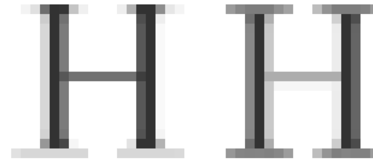

Figure 10. Times New Roman 'H' rendered from an ADF without (left) and with (right) alignment of the bottom-left corner of the glyph with the pixel grid.

Figure 11. Helvetica 'n' rendered from an ADF without (left) and with (right) automatic hinting of left and bottom edges by darkening pixels where the gradient of the distance field indicates a left or bottom edge and lightening pixels corresponding to right and top edges to maintain stroke weight.

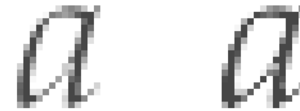

Figure 12. Palatino 'a' rendered from an ADF without (left) and with (right) automatic hinting to thicken thin strokes by slightly darkening pixels where the gradient of the distance field changes abruptly between adjacent pixels.

outline and fills it or b) defines a path stroked by a pen nib with numerous attributes, and 3) component-based design tools, which allow designers to build basic components such as stems, arcs, and other recurring shapes and then combine the components to create glyphs. See [Shamir and Rappoport 1998; Hu and Hersch 2001] for discussions on the relative merits and drawbacks of these methods.

We have built upon prior work in 3D sculpting with ADFs [Perry and Frisken 2001] to create a 2D font editor. This editor provides:

- Stroke-based design – the 2D counterpart to 3D carving in [Perry and Frisken 2001]. Stroking can be done interactively or it can be scripted to emulate programmable design tools.
- Curve-based design – using Bezier curve manipulation modeled after Adobe Illustrator combined with methods for converting outlines to ADFs (described in Section 3.3.2) and ADFs to outlines (described below)
- Component-based design – uses CSG and blending operations on the implicit distance field, allowing components to be designed separately and combined either during editing or during rendering (when using component-based compression as described in Section 3.3.3)
- A method for automatically generating ADFs from analog and digital font masters

Building the editor required several advances over the system described in [Perry and Frisken 2001]. For component-based design, we added the ability to efficiently reflect and rotate ADFs using quadtree manipulation described in [Samet 1989] to model the symmetries common in glyphs. Additional features include ADF scaling, translation, and operations to combine multiple ADFs (e.g., CSG and blending).

For stroke-based design, we added carving tools with a rectangular profile to emulate pen nibs. The orientation and width of the pen nib can change along the stroke to mimic calligraphy. During the stroke, we generate an ordered list of points along the stroke's path and then apply the curve fitting algorithm of Schneider [1990] to fit a minimum set of $G^2$ continuous curves to

the path (with user-specified accuracy). We then generate two additional ordered lists of offset points from this path using the tool width and orientation and fit curves to these offset points to generate the stroke outlines. The outline curves are placed in a spatial hierarchy for efficient processing [Johnson and Cohen 1998]. Finally, we generate the 2D ADF from this hierarchy using the tiled generator described in [Perry and Frisken 2001]. The minimum distance to the outlines is computed efficiently using Bezier clipping [Sederberg and Nishita 1991]. On a 2GHz Pentium IV processor, this approach is fast enough so that when converting from strokes to ADFs there is no perceptual delay for the user.

For curve manipulation, we added a Bezier curve editor [Farin 2001] modeled after Adobe Illustrator. We also added the ability to convert ADFs to curves to provide a seamless interface between all three design paradigms. This conversion marches through leaf cells using the ADF hierarchy for fast neighbor searching [Frisken and Perry 2003], generates an ordered list of points along the zero-valued iso-contours of the ADF, and then fits curves as above. Unlike Schneider, who computes the curve error from the list of points, we compute the curve error directly from the ADF distance field. We pay special attention to corners using the method described in [Itoh and Ohno 1993]. This approach is fast enough to allow users to switch between paradigms without any noticeable delay.

To incorporate the existing legacy of fonts stored in non-digital form (i.e., as *analog masters*) or in digital form as bitmaps (i.e., as *digital masters*), our editing system provides a method for generating ADFs from high-resolution bi-level bitmaps. Analog masters are first scanned to produce bi-level digital masters at a resolution at least 4x higher than the target ADF resolution (e.g., a 4096 x 4096 digital master is adequate for today's display resolutions and display sizes). An exact Euclidean distance transform [Maurer et al. 2001] is then applied to the bitmap to generate a regularly sampled distance field representing the glyph. We then generate the ADF from this regularly sampled distance field using the tiled generator of [Perry and Frisken 2001]. Conversion from the bitmap to the ADF requires ~10 seconds per glyph on a 2GHz Pentium IV processor.

### 6 A Computational Substrate for Kinetic Typography

The distance field and the spatial hierarchy of the ADF framework proposed in this paper play an important role in simulation. For example, both components can be used in collision detection and avoidance, for computing forces between interpenetrating bodies, and for modeling soft body deformation [e.g., Cani 1998; Desbrun and Gascuel 1995]. Level set methods [Osher and Sethian 1988; Sethian 2001; Osher and Fedkiw 2002] make use of signed distance fields to model numerous effects such as melting and fluid dynamics. ADFs are a compact implicit representation which can be efficiently queried to compute distance values and gradients, two important computations required for the methods listed above. In contrast, determining distance values and gradients from outlines that are moving or deforming is impractical in software for real-time interaction [Hoff et al. 2001]. Hoff et al. use graphics hardware to generate a regularly sampled 2D distance field on the fly for deforming curves approximated by line segments.

Figures 13–15 are still images from short animated vignettes provided on the accompanying webpages. Each vignette demonstrates some of the ways in which ADFs can be used to create interesting effects for kinetic typography. In Figure 13, the distance field is used to model real-time soft body deformations following Cani [1998]. In Figure 14, the distance field is exploited in a real-time particle system for attraction, repulsion, and collision detection. The implicit nature of the distance field permits complex topological changes (such as the surface offsets shown in Figure 14) that would be difficult to model with outline-based fonts. Figure 15 demonstrates the deformation of a glyph due to a vorticity field using level set methods. The distance field, used in the level set modeling, is also used to provide non-photorealistic rendering of the evolving glyph to add artistic effect.

Figure 13. A single frame from an animated vignette to illustrate the use of ADFs to model real-time soft body deformation.

Figure 14. Three frames from an animated vignette which uses the ADF to attract and repel particles in a real time particle system and uses the implicit nature of the distance field to create offset surfaces from the glyphs. The distance field permits complex topological changes that would be difficult to model with outline-based fonts.

## 7 Conclusions and Future Work

We have proposed a new framework for representing, rendering, editing, and animating character glyphs that uses 2D bi-quadratic ADFs. The bi-quadratic reconstruction function provides an optimal balance between memory use and computational load. The distance field provides an effective pre-filtering of glyph outlines which provides better antialiasing using a single unhinted sample per pixel than the supersampling methods used in the industry. The spatial hierarchy of the ADF permits efficient optimal adaptive supersampling for superior temporal antialiasing. The distance field also provides a computational substrate for automatic hinting, for unifying three common digital font design paradigms, and for creating a variety of special effects for kinetic typography.

There are many avenues of research to be pursued. Some of particular interest include performing rigorous perceptual studies, a full treatment of automatic hinting, extending this framework to exploit the repeating pattern of addressable colored sub-pixels in LCD displays as is done in ClearType, and using distances and distance gradients for rendering motion blur.

## 8 References


ADOBE SYSTEMS, INC. 1990. *Adobe Type 1 Font Format*. Addison-Wesley.

AKENINE-MOLLER, T. AND HAINES, E. 2002. *Real-Time Rendering*. A.K. Peters, Natick, MA.

APPLE COMPUTER, INC. 1990. TrueType Reference Manual. Available at developer.apple.com/fonts.

BETRISEY, C., BLINN, J. F., DRESEVIC, B., HILL, B., HITCHCOCK, G., KEELY, B., MITCHELL, D. P., PLATT, J. C. AND WHITTED, T. 2000. Displaced Filtering for Patterned Displays. In *Proc. Society for Information Display Symp.* pp. 296-299.

BLOOMENTHAL, J. 1997. *Introduction to Implicit Surfaces*, Morgan Kaufman.

CANI, M-P. 1998. Layered Deformable Models with Implicit Surfaces. In *Proc. Graphics Interface'98*. pp. 201-208.

CHO, P. 1999. Pliant Type: Development and Temporal Manipulations of Expressive, Malleable Typography. Master's Thesis, Media Arts and Sciences, MIT.

COOK, R. 1986. Stochastic Sampling in Computer Graphics. *In ACM Transactions on Graphics*, pp. 51-72.

DESBRUN, M. AND GASCUEL, M-P. 1995. Animating Soft Substances with Implicit Surfaces. In *Proc. SIGGRAPH 1995*. pp. 287-290.

FARIN, G. 2001. *Curves and Surfaces for CAGD: A Practical Guide*. Morgan Kaufmann.

FRISKEN, S., PERRY, R., ROCKWOOD, A. AND JONES, T. 2000, Adaptively Sampled Distance Fields: a General Representation of Shape for Computer Graphics. In *Proceedings ACM SIGGRAPH 2000*, pp 249-254.

FRISKEN, S. AND PERRY, R. 2002. Efficient Estimation of 3D Euclidean Distance Fields from 2D Range Images. In *Proc. IEEE/ACM SIGGRAPH Volume Visualization and Graphics Symposium 2002*, pp. 81-88.

FRISKEN, S. AND PERRY, R. 2003. Simple and Efficient Traversal Methods for Quadtrees and Octrees. To appear in *Journal of Graphics Tools*. See also MERL technical report TR2002-41.

GUPTA, S. AND SPROULL, R. 1981. Filtering Edges for Grayscale Displays. In *Computer Graphics* 15(3), pp. 1-5.

HERSCH, R. 1993. *Visual and Technical Aspects of Type*. Cambridge University Press.

HERSCH, R. 1987. Character Generation Under Grid Constraints. In *Proceedings ACM SIGGRAPH 1987*, pp. 243-252.

HERSCH, R., BETRISEY, C., BUR, J. AND GURTLER A. 1995. Perceptually Tuned Generation of Grayscale Fonts. In *IEEE CG&A*, Nov, pp. 78-89.

HERZ, J. AND HERSCH, R. 1994. Towards a Universal Auto-hinting System for Typographic Shapes. *Electronic Publishing,* 7(4), pp. 251-260.

HOFF, K., ZAFERAKIS, A., LIN, M. AND MANOCHA, D. 2001. Fast and Simple 2D Geometric Proximity Queries Using Graphics Hardware. In Proc. Interactive 3D Graphics'01.

HU, C. AND HERSCH, R. 2001. Parameterizable Fonts Based on Shape Components. In *IEEE CG&A* May/June, pp. 70-85.

ITOH, K. AND OHNO, Y. 1993. A Curve Fitting Algorithm for Character Fonts. In Electronic Publishing 6(3), pp. 195-205.

JOHNSON, D. AND COHEN, E. 1998. A Framework For Efficient Minimum Distance Computations. In *Proc. IEEE International Conference on Robotics and Automation*, pp. 3678-3684.

JONES, T. AND PERRY, R. 2000. Antialiasing with Line Samples. In *Proceedings Eurographics Rendering Workshop*, pp. 197-205.

JOY, K., GRANT, C., MAX, N. AND HATFIELD, L. 1998. Chapter 8: Anti-aliasing Algorithms. In *Tutorial: Computer Graphics: Image Synthesis*. Computer Soc. Press, Washington DC.

KNUTH, D. 1979. *TEX and METAFONT: New Directions in Typesetting*. Digital Press, Bedford, MA.

LANCASTER, P. AND SALKAUSKAS, K. 1990. *Curve and Surface Fitting: An Introduction.* Academic Press, San Diego, CA, (pp. 145-147, pp. 157-163) 1990.

LEE, J., FORLIZZI, J. AND HUDSON, S. 2002. The Kinetic Typography Engine: An Extensible System for Animating Expressive Text. In *Proc. UIST'02*, pp. 81-90.

LEE, M., REDNER, R. AND USELTON, S. 1985. Statistically Optimized Sampling for Distributed Ray Tracing. In *Proceedings ACM SIGGRAPH 1985*, pp. 61-67.

MACROMEDIA. 2003A. Macromedia Flash Technotes: Using Fonts in Flash 5 and Later. Available at www.macromedia.com /support/flash.

MACROMEDIA. 2003B. Macromedia MX: Delivering Effective User Interfaces. Available at www.macromedia.com/software /mx/info/effective_user_exp.html.

MAURER, C., RAGHAVAN, V. AND QI, R. 2001. A Linear Time Algorithm for Computing the Euclidean Distance Transform in


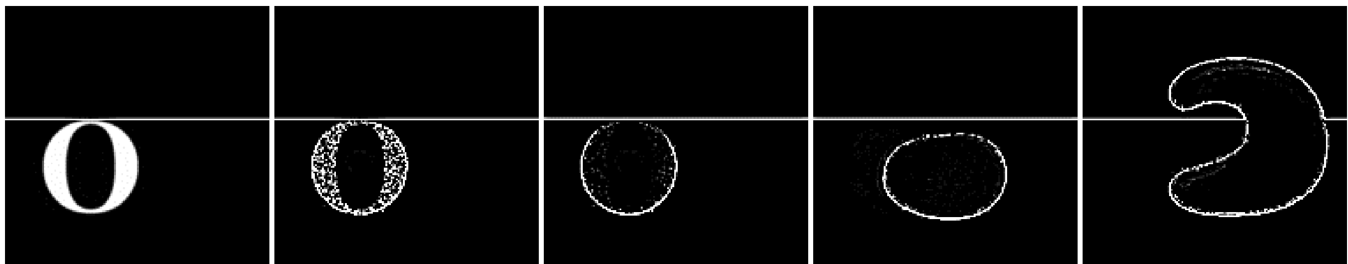

Figure 15. Multiple frames from an animated vignette which uses the distance field during level set modeling to evolve the silhouette of a glyph at interactive rates. The distance field is also used to provide non-photorealistic rendering of the evolving glyph. The glyph undergoes topological changes due to a vorticity field centered in the scene.

Arbitrary Dimensions. In *Proc. IPMI 2001*, pp. 358-364.
MAX, N. 1990. Antialiasing Scanline Data. In IEEE CG&A, 10(1) pp. 18-30.
MCNAMARA, R., MCCORMACK, J. AND JOUPPI, N. 2000. Prefiltered Antialiased Lines Using Half-Plane Distance Functions. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware 2000*, pp. 77-86.
MICROSOFT. 2002. What is ClearType. White paper available at www.microsoft.com/typography/cleartype/what.html.
MITCHELL, D. 1987. Generating Antialiased Images at Low Sampling Rates. In *Proc. ACM SIGGRAPH 1987*, pp. 65-72.
MITCHELL, D. 1996. Consequences of Stratified Sampling In Graphics. In *Proc. ACM SIGGRAPH 1996*, pp. 277-280.
MITCHELL, D. AND NETRAVALI, A. 1988. Reconstruction Filters in Computer Graphics. In *Proc. ACM SIGGRAPH 1988*, pp. 221-228.
O'REGAN, K., BISMUTH, N., HERSCH, R. AND PAPPAS, A. 1996. Legibility of Perceptually-Tuned Grayscale Fonts. In *Proc. IEEE Int. Conf. Image Processing*, pp. 537-540.
OSHER, S. AND FEDKIW, R. 2002. *Level Set Methods and Dynamic Implicit Surfaces*. Springer Verlag.
OSHER, S., AND SETHIAN, J. 1988. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton – Jacobi Formulations, *Journal of Computational Physics*, pp. 12-49.
PERRY, R. AND FRISKEN, S. 2001. Kizamu: A System for Sculpting Digital Characters. In *Proceedings ACM SIGGRAPH 2001*, pp. 47-56.
PLATT, J. 2000. Optimal Filtering for Patterned Displays. In *IEEE Signal Processing Letters*, 7(7), pp. 179-180.
RUEHLE, G. AND HALFORD, G. 1999. Method and Apparatus for Displaying Anti-Aliased Text. US Patent No. 5,940,080.
SAMET, H. 1989. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.
SEDERBERG, T. AND NISHITA, T. 1991. Geometric Hermite Approximation of Surface Patch Intersection Curves. In *CAGD*, 8(2), pp.97-114.
SETHIAN, J. 1999. *Level Set Methods and Fast Marching Methods : Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press.
SHAMIR, A. AND RAPPOPORT A. 1998. Feature-based Design of Fonts Using Constraints. In *Proc. Electronic Publishing 1998*, pp. 93-108.
SCHNEIDER, P. 1990. An Algorithm for Automatically Fitting Digitized Curves. In *Graphics Gems I*, ed. A. Glassner, pp. 612-626.
SMALL, D. 1985. Expressive Typography. Master's Thesis, Media Arts and Sciences, Massachusetts Institute of Technology.
SRAMEK, M. AND KAUFMAN, A. 1999. Alias-Free Voxelization of Geometric Objects. In *IEEE Transactions on Visualization and Computer Graphics,* 3(5), pp. 251-266.
THOMAS, B. 2002. Font Fusion: Technology for Making Text Look Good Anywhere. White paper available at www.bitstream.com.
TURKOWSKI, K. 1982. Anti-Aliasing Through the Use of Coordinate Transformations. In *ACM Trans on Graphics*, 1(3) pp. 215-234.
WESTOVER, L. 1990. Footprint Evaluation for Volume Rendering. In *Proceedings ACM SIGGRAPH 1990*, pp. 367-376.
WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. In *Communications of the ACM*, 23(6), pp. 343-349.
WOLBERG, G. 1994. *Digital Image Warping*. IEEE Computer Soc. Press, Los Alamitos, CA.
WONG, Y. 1995. Temporal Typography: Characterization of Time-Varying Typographic Forms. Master's Thesis, Medial Arts and Sciences, Massachusetts Institute of Technology.
ZONGKER, D., WADE, G. AND SALESIN, D. 2000. Example-Based Hinting of TrueType Fonts. In *Proceedings ACM SIGGRAPH 2000*, pp. 411-416.