A Resolution-Independent Representation for Pen-and-Ink Illustrations

Mike Salisbury Corin Anderson
Dani Lischinski David H. Salesin

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

Technical Report UW-CSE-96-01-02
January 1996

Abstract

This paper describes a compact resolution- and scale-independent representation for pen-and-ink illustrations. The proposed representation consists of a low-resolution grey-scale image, augmented by a set of discontinuity segments. We also present a new reconstruction algorithm that magnifies the low-resolution image while keeping the image sharp along the discontinuities. By storing pen-and-ink illustrations in this representation, we can produce high-fidelity illustrations at any scale and resolution by generating an image of the desired size and filling that image with pen-and-ink strokes.

iballs.ps

1 Introduction

The medium of pen and ink offers many advantages for visually communicating ideas. Pen-and-ink illustrations can be easily printed alongside text, using the same ink on the same paper, without degradation. Moreover, good reproduction quality can be obtained on commonplace 300 or 600 dot-per-inch laser printers, as well as on lower-resolution monochrome displays. Although pen-and-ink illustrations allow only monochromatic strokes of the pen, the resulting illustrations are often striking in their beauty and simplicity [10, 24].

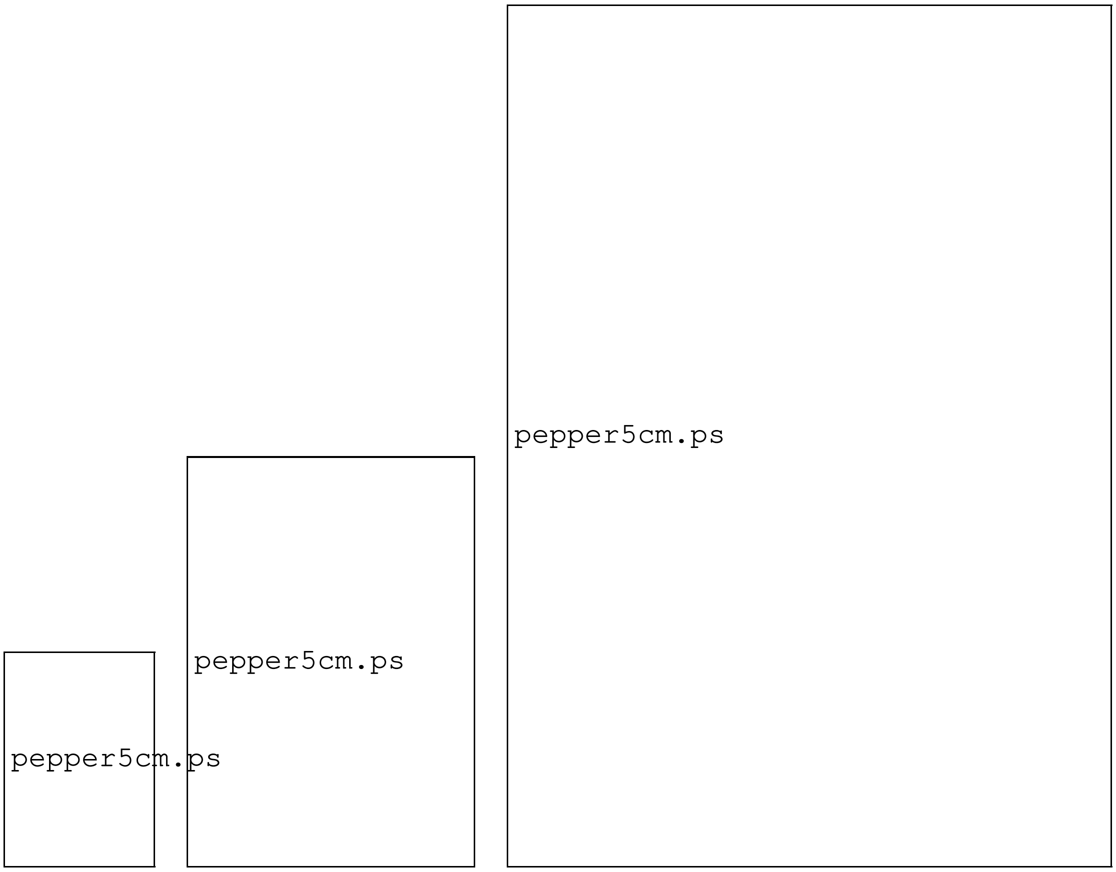
While the areas of photorealistic rendering and paint systems have received considerable attention in the literature, creating pen-and-ink illustrations on a computer is a relatively new area. Recently, Winkenbach and Salesin [27] described an automated rendering system that produces pen-and-ink illustrations from 3D polyhedral architectural models. This system employs an analytic visible surface algorithm to compute a planar map of the model's projection onto the image plane. Each region in the map can then be filled with the procedural stroke texture assigned to the corresponding 3D surface. Thanks to the analytic planar map and the procedural textures, the system can render the same view of the model at different scales and resolutions. While the different renderings generally employ different collections of individual strokes, the overall effect of the illustration is preserved.

Concurrently, Salisbury *et al.* [23] proposed an interactive system that allows the user to "paint" an image with prioritized stroke textures. This system is particularly useful for applying stroke textures to a scanned or synthetic image, effectively creating an "artistically-halftoned" monochrome version of the original. The user creates the pen-and-ink illustration on the screen, and the illustration is saved as a long list of Bézier strokes. One problem with this straightforward WYSIWYG approach is that illustrations represented in this manner cannot be reproduced at different scales or resolutions without significantly changing their overall appearance.

By way of example, consider the three illustrations shown in Figure 1. Figure 1(b) shows the original illustration at the size for which it was designed by the artist. If we wish to double the size of the illustration, we cannot merely scale it by a factor of two; such a scaling lightens the tone by spreading the same number of strokes over a larger area, as demonstrated by Figure 1(c). While this effect could be mitigated by thickening the strokes in the scaled-up version, the character of the illustration would be considerably altered. Conversely, scaling the illustration down darkens the tone, as the density of strokes increases (Figure 1(a)). Changing resolutions could also produce unwanted effects. For instance, all three illustrations would look darker on a 300 dot-per-inch printer, as they use strokes that are too thin for that resolution.

Another problem with storing illustrations as Bézier strokes is that the sheer number of strokes can make such a representation expensive to store, slow to transmit, and cumbersome to include in electronic documents. For example, the size of each of the three illustrations in Figure 1 is about one megabyte in PostScript [1].

In this paper, we extend the work of Salisbury *et al.*, by proposing an alternative representation for pen-and-ink illustrations that is scale- and resolution-independent, as well as compact. Instead of



(a) (b) (c)

Figure 1 The same illustration at three different scales. (b) is the original. (a) and (c) demonstrate that traditional rescaling changes tone and character.

storing each of the individual strokes comprising an illustration, we keep an underlying grey-scale image for each stroke texture in the illustration, along with a pointer to the stroke texture itself. To render the illustration at a particular scale and resolution, the grey-scale images are first rescaled and then hatched with strokes.

The proposed image-based representation is quite simple; however, maintaining true scale- and resolution-independence also requires solving an interesting related problem whose solution is not so straightforward. Since pen-and-ink illustrations hatch an image with strokes, they tend to be insensitive to very small image details. Thus it is often sufficient for the underlying grey-scale image to have a relatively low resolution. However, magnifying a low-resolution image for reproducing a large illustration on a high-resolution output device (such as a 600 dpi printer) typically results in undesirable blurring of the hard edges, or *discontinuities*, in the image.

The question, then, is how to resample images while preserving certain discontinuity edges. In order to be able to produce crisp edges in illustrations at all possible output scales and resolutions, we need to maintain information about discontinuity edges in the underlying image and explicitly take these edges into account in the resampling process. In this paper, we describe a new resampling algorithm to implement this process. This algorithm, which is essentially a discontinuity-sensitive reconstruction filter, is the main technical contribution of the paper.

1.1 Related work

Line-art illustration has been explored previously by a number of authors. Elber [9], Saito and Takahashi [21], Winkenbach and Salesin [27], and Lansdown and Schofield [13] all use 3D models to generate illustrations. Pneuli and Bruckstein [20] and Salisbury *et al.* [23] both generate illustrations starting from grey-scale images. However, neither of these works addresses the problem of rescaling images re-rendered from grey-scale images.

There are several paint systems that offer a measure of resolution-independence. Paint By Numbers [11] stores images as collections of resolution-independent strokes, Live Picture [15] represents the operations on images as a resolution-independent history, and LivePaint [19] provides resolution-independent procedural ink. However, none of these approaches provides any means of magnifying scanned images beyond their original resolution while preserving discontinuities. The resampling algorithm described in this paper could conceivably be used for this purpose in any of these systems.

The idea of making explicit use of discontinuities in functions and in images is not new: discontinuities have been used to construct good meshes for radiosity [12, 14] and to fit piecewise-cubic interpolants for radiance functions [22]. Yomdin and Elichai [29] also describe an image compression algorithm that locates and utilizes various types of edges in images to obtain a lossy compression scheme that avoids reconstruction artifacts in the vicinity of edges.

1.2 Overview

In the next section we describe in more detail our proposed representation scheme for pen-and-ink illustrations, and we present a new algorithm for image rescaling that preserves the discontinuities of the original image. Section 3 describes how pen-and-ink illustrations are created and viewed in our illustration system. Section 4 presents several examples of illustrations and describes our experience with the proposed representation. We conclude and offer directions for further research in Section 5.

2 Reconstructing images with discontinuities

This section describes the core of our pen-and-ink illustration system: an image-based representation for illustrations, and a new reconstruction algorithm for resampling images with discontinuities.

We have two major requirements of our representation. First, it should allow us to produce pen-and-ink illustrations at any scale and resolution, without changing the tone or character of the illustration.

Second, the resulting illustrations must keep "sharp features" sharp and "smooth features" smooth. By "sharp features" we mean abrupt changes in intensity along certain prescribed boundaries of the image, which we refer to as *discontinuities*. While we want our illustrations to exhibit crisp edges along discontinuities, we would like the tone to change smoothly everywhere else. In particular, it is important that the rescaling algorithm not introduce any spurious discontinuities.

One could imagine several possible representations that would meet our requirements. For example, we could maintain a history of all the operations performed on an image, along the lines of Live Picture [15], and then simply replay the history at the desired resolution when rendering the output. Although this approach is simple and basically sound, it has two main disadvantages. First, the representation's size and rendering time grow with the number of editing operations, and not necessarily with the complexity of the image. Second, this approach does not allow scanned or rendered images to be magnified beyond their original sizes without blurring their sharp features.

Another representation worth considering is storing the image as a collection of geometric patches covering the image area, with the heights of the patches representing the image color. One difficulty with this representation is the problem of handling discontinuities, since it is not obvious how to modify a smooth patch representation to be discontinuous along arbitrary curves. Another difficulty is the problem of determining control points so that the surface accurately approximates an arbitrary target image without introducing any ringing artifacts from maintaining the smoothness constraints.

For our representation, we have chosen to use a combination of uniformly-spaced image samples and piecewise-linear discontinuity curves called *discontinuity edges*. Any configuration of non-intersecting discontinuity edges is allowed. Thus, discontinuity edges are not constrained to form closed regions. Instead, we allow "open" discontinuities that have dangling edges not connected to any other. We would like the tone to change smoothly around the open end of such a discontinuity, but change sharply across the discontinuity edge. It is crucial to allow open discontinuities, as they frequently arise in images, especially when discontinuity edges are obtained by performing edge detection on a scanned image (see Figure 7(a)).

In the rest of this section, we'll describe the algorithm we use to produce an image of arbitrary scale and resolution from our representation.

2.1 Problem statement

The problem that we would like to solve can be stated formally as follows:

Given: A set of uniformly-spaced discrete sample (pixel) locations $\{x_i\}$, a set of corresponding intensity values $\{f_i\}$, and a set of line segments $\{\ell_i\}$ (the discontinuity edges);

Find: A function $f(x)$ that is smooth everywhere except across the discontinuity edges, such that $f(x)$ interpolates the values f_i.

The reconstructed function $f(x)$ can then be resampled at a higher rate than the original image, re-

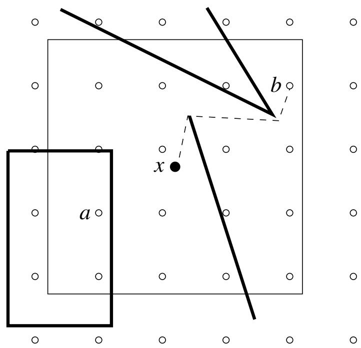


Figure 2 Several discontinuity edges (shown as thick lines) intersect the support of a 4×4 kernel centered at x. The dashed line indicates the shortest unobstructed path from x to b.

sulting in a magnified version of the image, or it can be resampled at a lower rate after band-limiting, yielding a minified version of the image.

2.2 Reconstruction algorithm

Reconstruction of a continuous signal $f(x)$ from uniformly-spaced discrete samples $\{(f_i, x_i)\}$ may be performed by convolving with a reconstruction kernel $k(x)$:

$$f(x) = \sum f_i \, k(x - x_i). \tag{1}$$

A variety of reconstruction kernels are available, including several interpolation kernels that effectively fit interpolants of various degrees to the input samples [28]. In this section, we describe a modification to the standard convolution framework that will cause the reconstructed function to be discontinuous across the discontinuity edges, while preserving smoothness everywhere else.

In order to reconstruct the value at point x, we first check whether there are any discontinuity edges that cross the support of the kernel centered at x. If no such discontinuities exist, the reconstructed value $f(x)$ is given by equation (1). Things become more interesting when one or more image samples under the kernel's support are separated from x by a discontinuity edge. Consider, for example, the situation in Figure 2. The thick solid lines indicate discontinuity edges that intersect the square support of the kernel centered at x (marked by a black dot). Input samples such as a cannot be reached from x without crossing a discontinuity edge. Clearly, such samples should not have any effect on the value of $f(x)$. Input samples such as b cannot be "seen" directly from x; however, they can be reached by going around the discontinuities. To ensure that $f(x)$ changes smoothly as x moves around these discontinuities, the sample b should have some effect on the value $f(x)$. Intuitively, this effect should be commensurate with the "ease" of reaching b from x without crossing discontinuities.

Thus, in order to preserve discontinuities in the reconstructed function $f(x)$, we replace the standard reconstruction kernel k with a modified kernel $\tilde{k}$, which attenuates k's entries according to each entry's reachability. To describe our new kernel, we must first define some terms.

Let $d(x, x_i)$ be the Euclidean distance between the two points, and let $sp(x, x_i)$ be the length of the shortest unobstructed path from x to x_i (see Figure 2). We explain how to compute the length of this path later in this section.

We define the *detour cost* between x and x_i as:

$$detour(x, x_i) = sp(x, x_i) - d(x, x_i) \tag{2}$$

Our modified kernel $\tilde{k}$ thus attenuates k:

$$\tilde{k}(x - x_i) = \alpha(x, x_i)\, k(x - x_i), \tag{3}$$

where the attenuation weighting function $\alpha(x, x_i)$ is defined as:

$$\alpha(x, x_i) = \begin{cases} 1 & \text{if } detour(x, x_i) = 0 \quad \text{(i.e., when } x_i \text{ is visible from } x\text{)} \\ 0 & \text{if } detour(x, x_i) \geq r \quad \text{(i.e., when } x_i \text{ is too "far" from } x\text{)} \\ 1 - 3t^2 + 2t^3 & \text{if } detour(x, x_i) < r, \quad \text{where } t = detour(x, x_i)/r \end{cases} \tag{4}$$

The constant r above is the detour cost beyond which no weight is given to a particular sample. We have found that a value of $r = 1$ works well for our 4×4 filter. The cubic polynomial in the third case above was chosen in order to ensure that $\alpha(x, x_i)$ is C^1 continuous.

Note that the new kernel $\tilde{k}$ no longer has the property that the sum of the weights corresponding to the samples under its support is one. To compensate, we turn the convolution in equation (1) into a *weighted-average* convolution:

$$f(x) = \frac{\sum f_i\, \alpha(x, x_i)\, k(x - x_i)}{\sum \alpha(x, x_i)\, k(x - x_i)}. \tag{5}$$

This form of convolution has been used previously for filtering non-uniform samples [7, 8, 17].

The kernel modification described above is applicable to any reconstruction kernel. In our implementation, we chose the cubic convolution kernel described by Wolberg [28]. This kernel has negative lobes, as do all interpolating kernels. This property introduces a slight complication into our reconstruction algorithm, as it is possible for the magnitude of the sum in the denominator of equation (5) to become very small. Dividing by small numbers magnifies any noise in the calculation, causing visible bright "speckles" in the reconstructed image. To overcome this difficulty, we switch to the entirely non-negative B-spline kernel [28] whenever the denominator in equation (5) falls below a certain threshold. To avoid introducing a discontinuity at places where the switch occurs, we smoothly blend between the two kernels as the denominator approaches zero. A precise definition of these two kernels is given in Appendix A.

Computing shortest paths

To complete the description of our reconstruction algorithm, it remains to explain how we compute the length of the shortest path between two points. We will refer to an endpoint of a discontinuity edge as a *discontinuity vertex*. If a discontinuity vertex is in the middle of a chain of discontinuity edges and is thus reachable from multiple sides, we consider each side of the vertex to be a distinct discontinuity vertex for the purposes of this minimum distance algorithm. As a preprocessing step, we compute the distance between all pairs of discontinuity vertices, using Dijkstra's all-pairs shortest paths algorithm [2]. Then, during reconstruction, to compute the shortest path between two particular points x_1 and x_2 that cannot "see" each other, we need to find the sets V_1 and V_2 of discontinuity vertices directly visible to each of these two points, respectively, within a certain ellipse that surrounds them both. This ellipse contains all points where the sum of the distances from the point to x_1 and x_2 is no larger than $d(x_1, x_2)$ plus r, the maximum detour cost from equation (4). Any discontinuity vertex beyond this distance would make the detour cost larger than r, thereby forcing the sample's weight $\alpha(x, x_i)$ to zero.

Our method for finding the sets V_1 and V_2 will be described shortly. Given these, the length of the shortest path between x_1 and x_2 is simply

$$sp(x_1, x_2) = \min_{v_1 \in V_1, v_2 \in V_2} \{d(x_1, v_1) + sp(v_1, v_2) + d(v_2, x_2)\},$$

where $d(\cdot, \cdot)$ is the Euclidean distance between two points, and $sp(v_1, v_2)$ is the length of the shortest path between vertices v_1 and v_2, as precomputed by Dijkstra's algorithm.

In order to rapidly determine V_1 and V_2 for any point x we first construct a constrained Delaunay triangulation (CDT) [6] containing all of the discontinuity edges in the image in the preprocessing step. Then, given a point x we locate the CDT face containing x. Starting from this face, we recursively visit all nearby faces that are reachable without crossing discontinuity edges. The vertices of all visited faces are tested to see if they are visible from x. To test the visibility of vertex v, we march from x towards v in the CDT and stop either when a discontinuity edge is crossed (in which case v is not directly visible), or when v is reached (in which case it is visible).

It costs $O(n^3)$ to compute the shortest paths between all pairs of n discontinuity edge vertices. The CDT can be constructed in $O(n \log n)$ time. Both of these computations are performed only once, in the preprocessing stage. For each point x at which the function is reconstructed (i.e., for each pixel location in the resampled image), we need to compute the detour cost for as many as sixteen pixels in the original image, as described above. The cost of this computation is no worse than quadratic in the number of discontinuity vertices within the kernel's support, although there are typically very few such vertices within this area.

3 Creating and printing illustrations

Each of our illustrations begins as a reference image. This image can be generated by rendering, digitally painting, or scanning in a printed image. We find discontinuity edges by using a Canny edge detector [5]. Then, to generate the image function sample values for our representation, we start with the imported image itself. For most pixels, this image is a good approximation to the sample values. But for pixels containing discontinuities, the image function takes on multiple values within the pixel and makes the average unsuitable as an approximation to the sample value. We therefore reconstruct these samples by extrapolating nearby valid samples from the same side of the discontinuity.

Once this process is completed, we reduce the size of the image as much as possible while maintaining enough detail to satisfy the user. If the sampling grid is made too sparse, some areas of the representation will have no reachable samples given their discontinuities. In this situation, we must obtain samples somehow, so we allow the reconstruction process to cross discontinuities. Thus, reducing the size of the image portion of our representation too much also has the effect of eliminating discontinuities.

Once the image has been reduced, we assign to it a stroke texture, along with an optional set of *outline strokes*, used to surround selected regions of the image, The outline strokes can be chosen from the set of discontinuity edges. To make use of multiple stroke textures, the image can be separated into grey-scale overlays, each of which is associated with its own stroke texture.

In order to view or print an illustration, we first produce a grey-scale image of the desired size with the algorithm described in Section 2. Since we want to avoid crisp, mechanical-looking edges in the final illustration, we magnify the image to one quarter of the desired size with our reconstruction algorithm, after which we expand the image by a factor of four using a standard reconstruction filter that ignores discontinuities. This trick generates enough blurring along the edges to give the illustrations a hand-drawn feel.

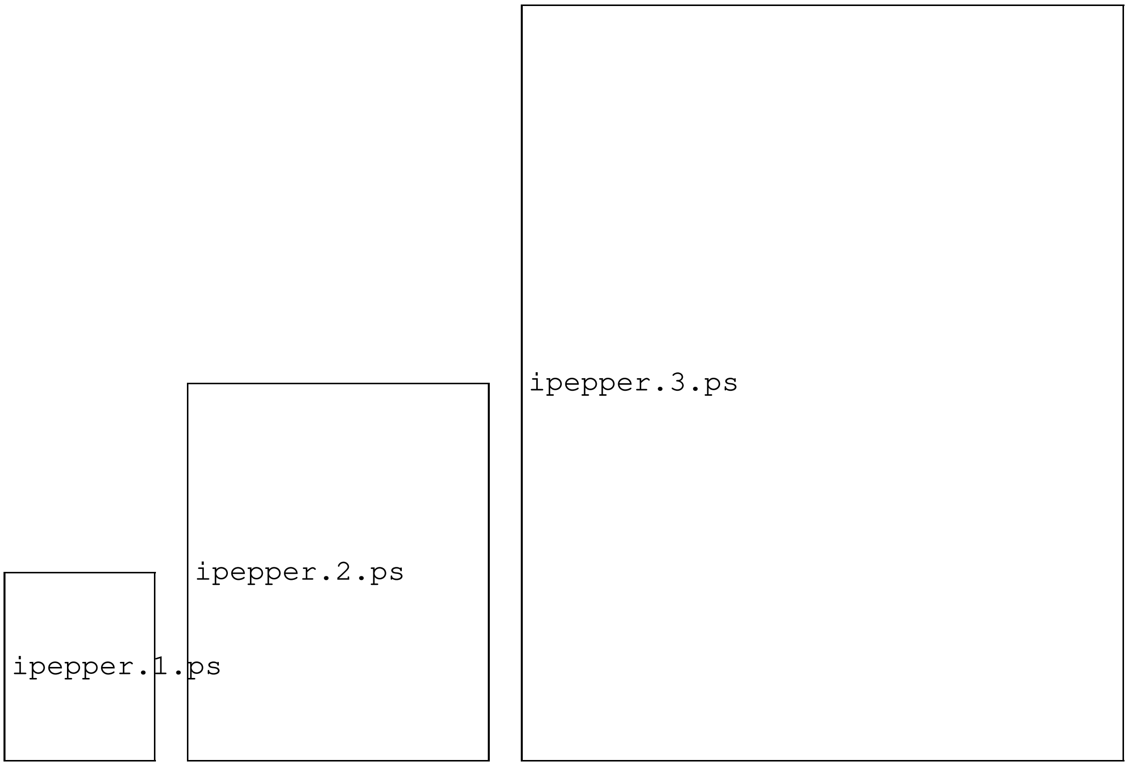
We then re-render the illustration with a prioritized stroke texture in an automatic process called *blasting*. Blasting creates an illustration with strokes, which, when taken together, produce the same tone that was present in the underlying grey-scale image. The blasting algorithm takes an image and a stroke texture as input. We use the same approach as Salisbury *et al.* [23] for producing strokes and placing them into an illustration. This approach consists of repeatedly selecting a stroke from the stroke texture and computing the darkness in the vicinity of the stroke that would result if it were added. The stroke is rejected if it makes the illustration darker than the tone of the underlying image anywhere along its length. Whereas the system of Salisbury *et al.* generates candidate strokes only underneath the user's brush, we automatically place strokes that cover the entire illustration.

4 Results

We present several illustrations that demonstrate the various capabilities of our representation.

Figures 3 and 4 show an illustration rendered at several scales. Each of these scales maintains the correct tone and gives the same overall texture effect. Note that different collections of actual strokes were used to generate the illustration at the various sizes. Compare these illustrations to those in Figure 1 where the same set of strokes was used for each scale, resulting in undesirable alterations in tone and overall effect.

Figure 5 demonstrates the advantage of maintaining and respecting discontinuities. Figure 5(a) is the illustration produced by blasting an image that was rescaled without maintaining discontinuities. The illustration in 5(b) is the same as 5(a) except that it was rescaled using our magnification algorithm. Notice that the outline edges do not align with the edges of the fingers in illustration 5(a). Also of particular interest is the use of multiple textures in both Figures 5 and 6.



(a) (b) (c)

Figure 3 A pepper at three different scales. Original photograph by Edward Weston [26].

The illustrations in Figure 7 are close-ups of the lower-right-hand corner of Figure 6 and show the potential for using our representation to generate poster-sized illustrations. Figure 7(a) shows the discontinuity edges used to generate 7(c) and, in turn, 7(e). The images in 7(b) and 7(c) are the grey-scale images that were used as references for illustrating the key's texture in 7(d) and 7(e). All of the edges in these and the other illustrations we present were found automatically with our edge detector.

Notice in Figure 7(a) that the key is not completely surrounded by edges. In places where no discontinuity edges are present, our reconstruction produces smooth changes in grey-scale, even in the vicinity of discontinuity endpoints.

Table 1 gives the storage requirements and reconstruction times required for the illustrations in this paper. The REPRESENTATION column gives the number of pixels stored on disk, the number of discontinuities used, and the total size of both in kilobytes. The OUTPUT column gives the size of the reconstructed grey-scale image that was used for blasting, and the size, in megabytes, of the PostScript file used for printing. Finally, the TIME column gives the time required to pre-process, enlarge, and blast the image with strokes. These times were measured on a 200MHz Indigo2. To summarize the table, our representation reduces the storage requirements of these illustrations by a factor of 100–1000, and it takes from 1–15 minutes to render them.

ipepper.4.ps

Figure 4 Enlarged pepper.

iegg-blurry.ps

(a)

iegg.ps

(b)

Figure 5 An illustration with standard filtering (a) and with our algorithm (b). Both illustrations were produced from images of the same resolution. Original photograph by Walter Swarthout [25].

5 Future work

Our experience with this representation suggests several areas for future research:

- *Algorithm optimizations.* Our algorithm is currently too slow to be considered interactive. One of the reasons magnification is slow is that we are performing a 2D filtering operation for each pixel in the output image that is affected by discontinuities. We are working on creating a separable filter that satisfies our output image criteria and that would allow us to perform a faster 1D magnification in each dimension. We have not yet investigated optimizing the CDT or blasting algorithms.

- *PostScript renderer.* The reduced size of our representation currently offers no practical advantage in transmission to printers or web browsers, because they do not recognize our representation. One solution is to write rendering code in the languages that these devices understand. For example, a PostScript printer could be sent illustration rendering code written in PostScript along with several standard stroke textures. This code could then generate an illustration at any requested scale and resolution directly on the output device. Similar programs could be written in Java or Acrobat to allow web browsers to render illustrations.

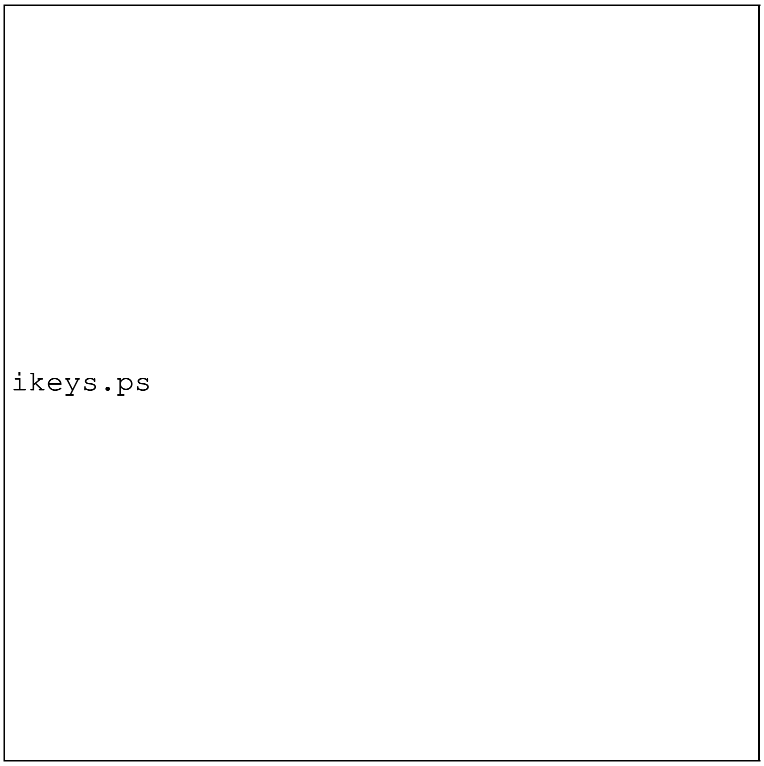


Figure 6 Keys. Original photograph by Randy Miller [16].

- *Curved discontinuity edges.* A current restriction of our algorithm is that we only support piecewise-linear discontinuity edges. We can approximate curved edges by a series of straight segments, but using too many segments can slow down our algorithm significantly. Allowing curved discontinuity edges would significantly reduce the number of discontinuities needed. The biggest challenge in implementing this change will be redesigning the shortest path computation in Section 2.2 to deal with curved segments.

- *Scalable textures.* Currently our stroke textures are applied at a single scale: if the illustration is magnified, the texture shrinks relative to the image. While this effect is acceptable for uniform textures, it could be objectionable for textures with recognizable patterns. It might be better to have a multiresolution stroke texture that could change scale with the illustration. Then, as the scale of the texture increased, finer resolution strokes could automatically be added to the illustration.

- *Combining with multiresolution images.* Although images with sharp detail can be generated at any resolution with our representation, the color resolution cannot currently be increased beyond the original scale due to the representation's fixed sampling grid. To be able to add higher-resolution color detail to our representation would require replacing our fixed-resolution image with a multiresolution image such as the one described by Berman *et al.* [4]. In this case, we would also want to develop a multiresolution discontinuity representation in which discontinuities could appear and disappear at different scales.

dkeys-closeup.ps

(a)

gskeys-closeup-blurry.ps

(b)

gskeys-closeup.ps

(c)

ikeys-closeup-blurry.ps

(d)

ikeys-closeup.ps

(e)

Figure 7 Close-ups of Figure 6. (a) shows the discontinuity edges used. (b) is the grey-scale image produced using a standard magnification filter. (c) used (a) to maintain sharp edges. (d) and (e) are illustrated from (b) and (c).

Fig	Content	REPRESENTATION ImgSize (pixels)	# Edges	Storage (KB)	OUTPUT ImgSize (pixels)	PS Size (MB)	TIME PreProc (sec)	Enlarge (sec)	Blast (sec)
	Billiards	67×96	326	5.4	568×488	1.3	68	81	78
3a	Pepper	102×128	270	10.5	204×256	0.2	61	3	14
3b	Pepper	102×128	270	10.5	408×512	1.0	61	3	46
3c	Pepper	102×128	270	10.5	816×1024	3.8	61	79	182
4	Pepper	102×128	270	10.5	1632×2048	15.2	61	242	717
5a	Hand	64×64	—	1.2	1024×1024	1.0	—	3	209
"	Egg	128×128	—	1.4	1024×1024	1.0	—	4	388
5b	Hand	64×64	237	2.3	1024×1024	1.0	30	195	213
"	Egg	128×128	143	1.9	1024×1024	1.0	6	123	393
6	Keys	128×128	504	7.9	1024×1024	2.4	317	137	218
"	Shadow	64×64	111	1.5	1024×1024	2.4	4	68	383
7d	Key closeup	27×22	—	0.7	864×704	1.9	—	2	130
"	Shadow	27×22	—	0.2	864×704	1.9	—	2	250
7e	Key closeup	27×22	107	1.0	864×704	2.0	4	226	130
"	Shadow	27×22	6	0.3	864×704	2.0	1	16	255

Table 1 Illustration sizes and speeds.

- *Progressive image generation.* Other sample-based image synthesis techniques might also benefit from a representation that maintains and respects discontinuities. For example, when creating an image by adaptive ray casting, in which new rays are cast where the variance of the image is largest, most of the rays will be cast near the edges of objects and shadows [18]. However, if we instead determined the locations of edges as part of a pre-process and then used those edges in interpolating the ray-cast samples, it should be possible to create images of comparable quality with fewer rays or images of higher quality with the same number of rays.

- *Image compression.* Given our algorithm's ability to reproduce large images from a small representation, it is natural to consider the possibility of using it as a more general image compression mechanism. One complication with such an approach may be the lack of texture detail in the reconstructed images.

A Kernel definitions

The cubic convolution kernel we used is a member of the following family of kernels [28]:

$$k(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1 & 0 \leq |x| < 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a & 1 \leq |x| < 2 \\ 0 & 2 \leq |x| \end{cases} \tag{6}$$

We chose the the value $a = -0.5$, which makes the Taylor series approximation of the reconstructed function agree in as many terms as possible with the original signal.

The non-negative B-spline kernel we used is defined as follows:

$$k(x) = \frac{1}{6} \begin{cases} 3|x|^3 - 6|x|^2 + 4 & 0 \leq |x| < 1 \\ -|x|^3 + 6|x|^2 - 12|x| + 8 & 1 \leq |x| < 2 \\ 0 & 2 \leq |x| \end{cases} \tag{7}$$

This kernel is sometimes called the *Parzen window*.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, Reading, Massachusetts, 2nd edition, 1994.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1987.

[3] Arthur Beck. Billiard balls photograph. In *Photographis 81, The International Annual of Advertising and Editorial Photography*, page 151. Graphis Press Corp., 1981.

[4] Deborah F. Berman, Jason T. Bartell, and David H. Salesin. Multiresolution painting and compositing. In *Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 85–90. ACM SIGGRAPH, ACM Press, July 1994.

[5] John Canny. A computational approach to edge detection. In Rangachar Kasturi and Ramesh C. Jain, editors, *Computer Vision: Principles*, pages 112–131. IEEE Computer Society Press, Los Alamitos, California, 1991. Reprinted from *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[6] L. Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[7] Robert L. Cook. Stochastic sampling in computer graphics. *Transactions on Graphics*, 5(1):51–72, January 1986.

[8] Mark A. Z. Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. *Computer Graphics*, 19(3):69–78, July 1985.

[9] Gershon Elber. Line art rendering via a coverage of isoparametric curves. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):231–239, September 1995.

[10] Arthur L. Guptill. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York, 1976.

[11] Paul E. Haeberli. Paint by numbers: Abstract image representations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 207–214, August 1990.

[12] Paul Heckbert. Discontinuity meshing for radiosity. In *Third Eurographics Workshop on Rendering*, pages 203–226, Bristol, UK, May 1992.

[13] John Lansdown and Simon Schofield. Expressive rendering: A review of nonphotorealistic techniques. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

[14] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, November 1992.

[15] T. Marshall. Lively pictures (Power Mac image editing). *BYTE*, 20(1):171–172, January 95.

[16] Randy Miller. Keys photograph. In *Photographis 77, The International Annual of Advertising and Editorial Photography*, page 72. Graphis Press Corp., 1977.

[17] Don P. Mitchell. Generating antialiased images at low sampling densities. *Computer Graphics*, 21(4):65–72, July 1987.

[18] James Painter and Kenneth Sloan. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 281–288, July 1989.

[19] Ken Perlin and Luiz Velho. Live Paint: painting with procedural multiscale textures. In *Proceedings of SIGGRAPH 95 (Los Angeles, California, August 6–10, 1995)*, Computer Graphics Proceedings, Annual Conference Series, pages 153–160. ACM SIGGRAPH, ACM Press, August 1995.

[20] Yachin Pnueli and Alfred M. Bruckstein. $\mathbf{Dig^i}_D$*ürer* — a digital engraving system. *The Visual Computer*, 10(5):277–292, 1994.

[21] Takafumi Saito and Tokiichiro Takahashi. NC machining with G-buffer method. In Thomas W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 207–216, July 1991.

[22] David Salesin, Daniel Lischinski, and Tony DeRose. Reconstructing illumination functions with selected discontinuities. In *Third Eurographics Workshop on Rendering*, pages 99–112, Bristol, UK, May 1992.

[23] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 101–108. ACM SIGGRAPH, ACM Press, July 1994.

[24] Gary Simmons. *The Technical Pen*. Watson-Guptill Publications, New York, 1992.

[25] Walter Swarthout. Hand and egg photograph. In *Photographis 75, The International Annual of Advertising, Editorial, and Television Photography*, page 133. Graphis Press Corp., 1975.

[26] Edward Weston. *Aperture Masters of Photography, Number Seven*. Aperture Foundation, Inc., New York, New York, 1988.

[27] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100. ACM SIGGRAPH, ACM Press, July 1994.

[28] George Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, California, 1990.

[29] Y. Yomdin and Y. Elichai. Normal forms representation: a technology for image compression. In *Image and Video Processing*, volume 1903 of *Proceedings of the SPIE — The International Society for Optical Engineering*, pages 204–214. SPIE, February 1993.