Seventh Hungarian Conference on Computer Graphics and Geometry, Budapest, 2014

# Improving Texture-based NPR

László Szécsi

Marcell Szirányi

Tamás Umenhoffer

Budapest University of Technology and Economics, Budapest, Hungary

**Abstract**

*This paper presents a real-time procedural texturing algorithm for hatching parametrized surfaces. We expand on the concept of* Tonal Art Maps *to define self-similar, procedural tonal art maps that can service any required level-of-detail, allowing to zoom in on surfaces indefinitely. We explore the mathematical requirements arising for hatching placement and propose algorithms for the generation of the procedural models and for real-time texturing.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation

## 1. Introduction

Photo-realism has been in the focus of rendering systems for decades. Photo-realistic rendering aims at creating images that are indistinguishable from real-world photographs, which is made possible by the precise simulation of physics laws during the rendering process. How accurately physics is applied in the rendering algorithm determines the level of realism of the result.

Computer graphics also tries to mimic artistic expression and illustration styles[7, 16, 18]. Such methods are usually vaguely classified as *non photo-realistic rendering* (NPR). While the fundamentals of photo-realistic rendering are in optics that are well understood, NPR systems simulate artistic behavior that is not mathematically founded and often seems to be unpredictable. Therefore, the first step of NPR is to model the artist establishing a mathematical model describing his style, and then solve this model with the computer. The result will be acceptable if our model is close to the not formally specified artistic behavior. During the history of NPR, many individual styles were addressed. Hatching is one of the basic artistic techniques that is often emulated in stylistic animation.

Hatching strokes should appear hand-drawn, with roughly similar image-space width, dictated by pencil or brush size, but they should also stick to surfaces to provide proper object space shape and motion cues. Both properties must be maintained in an animation, without introducing temporal artifacts. We call these the requirements of image space and world space consistency. The two requirements are in contradiction when the relation of image space surface to object space surface is being altered, i.e. when surface distance or viewing angle is changing. The rendering process should resolve this contradiction while presenting natural randomness inherent in manual work[10, 1].

This paper presents a hatching style NPR rendering algorithm that can be implemented in real-time. The main scientific contributions are

- the introduction of *self-similar tonal art maps*,
- an algorithm for generation of *self-similar seed sets*,
- a single pass, real-time hatching algorithm using the seed sets, with automatic, procedural, continuous level-of-detail (Figure 1) .

The organization of the paper is as follows. In Section 2 we summarize the related previous work on NPR. Section 3 introduces the idea of Self-similar Procedural Tonal Art Maps. In Section 4, we derive the mathematical construct for the placement of hatching strokes that meets the self-similarity requirements. We discuss the interpretation of the model in Section 5, including the level-of-detail scheme and stroke sizing. Tone representation is added in Section 6. A

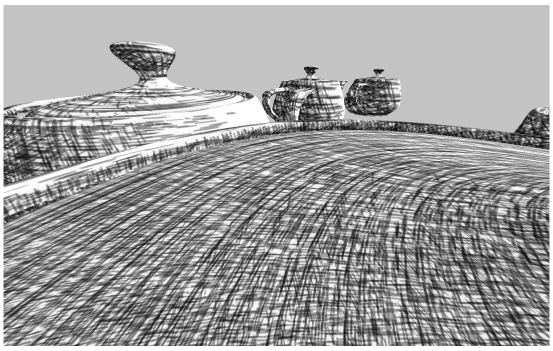

**Figure 1:** *Teapots rendered with the same shader and settings, featuring different levels of detail.*

detailed description of the final algorithm, and the discussion of results and future work conclude the paper.

## 2. Previous work

### 2.1. Density and direction fields

In pencil drawings artists convey the shape and illumination of objects with the density, orientation, length, width and shade of thin hatching strokes[23, 9]. To mimic this, we should find a *density* and a *direction field* in the image plane that is as close as possible to what an artist would use. The density, length, width and shade should be influenced by the current illumination, while the orientation is determined by the underlying geometry. Artists may use several layers of strokes, aligned at different angles to the direction field. If we defined the direction field directly in the image plane, it would be difficult to convey 3D shape or motion. Thus, it is beneficial to determine these directions on object surfaces either from geometric curvatures[6] or from the tone[13]. The *principal curvature directions*[6] of the surface define a field that is an intuitive representation of shape information.

In this paper, we do not address the problem of the curvature field generation, but assume that a proper UV parametrization is already known for surfaces, where isoparametric curves follow desired hatching directions. Proposing an algorithm tailored to specifically for our approach is left for future research.

### 2.2. Uniformly spaced and random hatching styles

In hand-made artwork, the artist may choose to pay attention to the distance of strokes and keep similar distance between them. The method of Zander[25] integrates the vector field to produce strokes also guaranteeing that they are separated which provides very high quality still images. Unfortunately, this kind of spatial control of strokes is in contradiction with temporal coherence. We have to accept that not all hatching styles are created equal regarding their applicability to animation. Specifically, to reproduce a given shade, we cannot apply strokes that are placed at strictly identical, fixed image space distances, and follow object space motion at the same time. When zooming in or changing the viewing angle, either one of the conditions must be broken. Violating object space consistency will cause the *shower door effect*, meaning the user has the impression that strokes are not fixed to objects but are floating on them. If we allow the regular hatching distance to change, coverage will deviate from the desired shade. The continuous level-of-detail feature of hatching is absent from these techniques.

Counterpointing the above considerations against overly regular hatching, it is also important that strokes are distanced evenly in a statistical sense, and that the distribution of distances between them does not depend on the direction. As strokes make the shading anisotropic, the distribution of stroke locations itself should not only be uniform but also isotropic. Otherwise, superimposed the two will result in direction-dependent patterns, where strokes clump together for unfortunate orientations.

### 2.3. Image space methods

To guarantee consistency with the image, hatching strokes can be generated directly in image space[11]. In order to avoid the shower door effect, strokes can be moved along with an optical flow or image space velocity field, but placing new strokes on emerging, previously non-visible surfaces still poses problems. Especially if strokes are long, following curvature or feature curves of object surfaces, they should maintain this even when only tiny fractions have become visible. This cannot be assured when only using image space information.

### 2.4. Seed-based methods

Several works[14, 20] proposed the application of *particles* or *seeds* attached to objects, but extruding them to hatching strokes in image space. Strokes are obtained by integrating the direction vector field started at seed points or particles[25, 15]. The key problem in these methods is the generation of the world-space seed distribution corresponding to the desired image-space hatching density. This either means seed killing and fissioning[24]—even using mesh subdivision and simplification[4]—, or rejection sampling[20]. These techniques are mostly real-time, but require multiple passes and considerable resources. Compositing hatching strokes with three-dimensional geometry is not straightforward: depth testing of extruded hatch curves against triangle mesh objects must be using heavy bias and smooth rejection to avoid flickering. While this allows for modeling some human inconsistencies in performing the same hidden line removal task, it is also extremely ponderous to eliminate them should they not be desired.

In our work, we use the notion of seeds to discuss hatching stroke position and distribution patterns. However, we are not concerned by seed positions in object space, rather in parameter space, and we do not extrude seeds to triangle strip geometry.

## 2.5. Texture-based methods

In order to make the pencil strokes consistent with the 3D objects, hatches can be generated into textures and mapped onto animated objects[12]. This requires surface parametrization. Unfortunately, this approach does not provide natural pencil art, where each stroke is drawn in image space. There, every stroke has roughly similar width which is determined by the pencil of the artist and is independent of the distance or the viewing angle of the depicted surface. From another point of view, the fact that pencil art is naturally produced in image space implicitly assumes a level-of-detail mechanism, which renders objects with fewer pencil strokes if they are farther away and thus cover just a smaller portion of the image[5].

The most characteristic limitation of texturing-based hatching approaches is limited level-of-detail support. Simple static textures perform extremely poorly, as the width of hatching strokes is fixed in UV space, and—through the UV mapping—also in object space. Note that changing the UV mapping depending on viewing distance is not only insufficient to address projection distortion, but also would result in the hatching pattern floating on the surface.

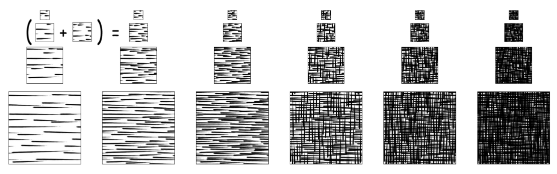

**Figure 2:** *A Tonal Art Map with the nesting property. Strokes in one image appear in all the images to the right and down from it. From Praun et. al*[16].

Thus, simple texturing does not allow for hatching that is uniform in screen space. A level-of-detail mechanism called *Tonal Art Maps* has been proposed to alleviate this problem[16]. Using this technique, several texture images are pre-drawn, representing different tones and hatching scales. Figure 2, taken from the referred paper, shows such a set of maps. When rendering surfaces, the appropriate texture in every pixel can be selected based on the desired tone and on-screen hatching stroke width. In order to avoid sharply clipped hatching strokes at boundaries of discrete zones, the patterns fade into each other using interpolation.

In an animation, as the required hatching density is changing, it is important that strokes stay at their on-surface positions. It is allowed for new hatching strokes to appear when the density increases, and for existing strokes to vanish, should the density decrease. However, strokes should not be appearing and vanishing in the same vicinity at the same time. Therefore, denser hatching textures should always contain the strokes of sparser textures as a subset. This is called the *nesting property*, also observable in Figure 2.

Tonal Art Maps, however, only support a range of hatching scales as defined by the most detailed and least detailed map levels. Thus, when zooming in onto a surface, we cannot have finer hatching than what texturing with the most detailed map level would produce, resulting in classic texture magnification artifacts, and huge and sparse hatching strokes in image space. Also, there is a trade-off between the number of detail levels used and the quality of transitions between those levels. With too few textures, a large number of strokes fade in at the same time, resulting in an image with non-uniform stroke weights. While this is acceptable in most cases, as weaker pencil strokes can possibly be used by artists, it is a limitation to the degree of screen-space uniformity we can achieve.

## 3. Self-similar Tonal Art Maps

Our idea is to make Tonal Art Maps infinitely loopable, by making the nesting property recursive. Hatching strokes are positioned at *seeds*. The texture is broken into four tiles, forming a $2 \times 2$ grid. In all four quarters, the seeds must be the subset of the complete seed set scaled down to fit the quarter. That way the seeds are nested in the pattern we get by repeating them twice along both axes (see Figure 3). Thus, when we zoom in to any of the quarters, it is possible to add new strokes re-creating the original most detailed hatching pattern, where the process can be restarted (see Figure 4). This is the *recursive nesting property* of the seed set, which we will define more formally in Section 4. Following the classic Tonal Art Map scheme, this would require four sequences of images, describing how the individual quarters evolve into the full hatching pattern. However, we will never actually create these images, but describe them procedurally, and use this extremely compact representation for rendering.

First, in Section 4, we deal with the problem of placing the hatching strokes to assure the recursive nesting property. We start by considering seed point placement only, addressing hatching stroke length and width in Section 5.

## 4. Self-similar seed generation

We need a set of seed points to place strokes at. Let the set of all seed points be $S = \{\mathbf{s}_0, \ldots, \mathbf{s}_i, \ldots, \mathbf{s}_{N-1}\}$, where $\mathbf{s}_i = (s_{iu}, s_{iv})$. These positions are defined in the *seed space*, the relation of which to the UV space we explore in Section 5.

Let us define the operator $\mathfrak{D}$ as follows:

$$\mathfrak{D}\mathbf{s} = (\operatorname{frac}(2s_u), \operatorname{frac}(2s_v)),$$

where the frac function yields the fractional part of a number.

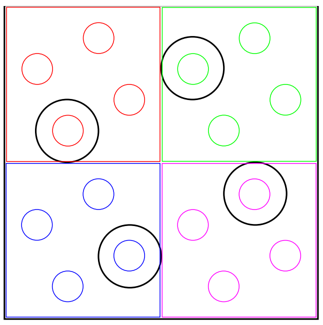

**Figure 3:** *The smallest possible seed set with the recursive nesting property. The four large circles indicate seeds, small circles are the seed pattern repeated on a $2 \times 2$ grid.*

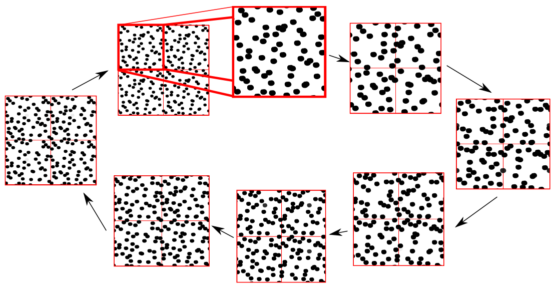

**Figure 4:** *The recursive nesting property ensures that by adding seeds to any of the quarters we can reproduce the original seed pattern. Seed markers decrease in size, but seed positions are unchanged.*

Geometrically, this operation is a scaling by the factor of two, using the nearest corner of the unit square as the pivot point (Figure 5). Numerically, if we consider the binary radix fraction form of the seed coordinates, the operation removes the first binary digit after the binary radix point, shifting the rest to the left (Figure 6). Note that in the unit square, there is always only a zero on the left side of the radix point.

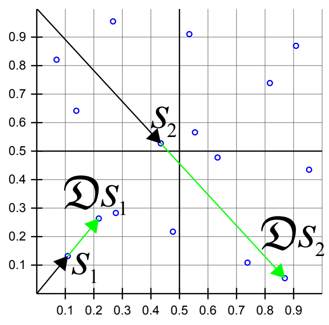

**Figure 5:** *Geometric interpretation of operator $\mathfrak{D}$. Seeds are projected to double their distance from the nearest corner.*

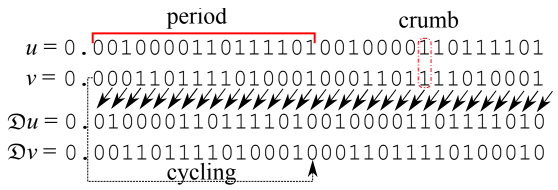
period
crumb
$u = 0.0010000110111110100100001110111101$
$v = 0.0001101111010001000110111111010001$
$\mathfrak{D}u = 0.0100001101111101001000011101111010$
$\mathfrak{D}v = 0.0011011110100010001101111101000010$
cycling

**Figure 6:** *Interpretation of operator $\mathfrak{D}$ on binary fractions. For periodic fractions, the bits can be cycled.*

The operator can be extended to sets as:

$$\mathfrak{D}S = \{\mathfrak{D}\mathbf{s} \mid \mathbf{s} \in S\}.$$

Set $S$ has the recursive nesting property if

$$\mathfrak{D}S \subseteq S.$$

This means that if there is a seed $\mathbf{s}_i$, then its image $\mathfrak{D}\mathbf{s}_i$ must also be a seed. A sequence of $N$ seeds then must be found as

$$\mathbf{s}_{i+1} = \mathfrak{D}\mathbf{s}_i, \text{ if } 0 \leq i < N,$$

because all seeds will trivially be mapped to another seed. However, $\mathfrak{D}\mathbf{s}_{N-1}$ must also be in $S$. This can be true if

$$\mathbf{s}_0 = \mathfrak{D}\mathbf{s}_{N-1}.$$

Considering the interpretation of $\mathfrak{D}$ on binary fractions, we can conclude that seeds must be generated by shifting the bit patterns of coordinates $s_{0u}$ and $s_{0v}$, and after $N$ steps we need to arrive back at $\mathbf{s}_0$. This is possible if $s_{0u}$ and $s_{0v}$ are periodic binary fractions of period $N$ (or a divisor of $N$). Such periodic binary fractions are easy to generate e.g. using the Bernoulli(1/2) process, finding individual bits as independent coin flips. Even though such a process generates points that are evenly distributed in the statistical sense, a concrete small set of seeds generated in such a way could be not filling the space evenly.

In an infinite random sequence, we expect any fixed-length subsequence to appear with exactly the same probability. Extending that to finite sequences, we expect possible fixed-length subsequences to appear with frequencies as uniform as possible when cycling through the sequence. Bit sequences with such a property are called *uniform cycles*[17]. Although no proof for the existence of arbitrary-length uniform cycles is known, all possible uniform cycles can be enumerated for modest lengths. The bit sequence for the $u$ and $v$ coordinates could be found independently, but then coincidentally identical substrings could appear. Instead, we can combine their respective bits to form *crumbs* (quaternary equivalent of binary bits or decimal digits), forming a quaternary periodic fraction. To distribute points evenly, the crumbs of one period must form a quaternary uniform cycle.

In order to understand what uniformity means in the geometrical sense, let us find the intuitive meanings of the crumb values. Recall that the first crumb in the quaternary representation of a seed point is the combination of the first bits of the binary fractions for its coordinates. Thus, the value of the first crumb indicates in which quarter of the unit square the seed point is. Then the second crumb indicates in which $\frac{1}{4} \times \frac{1}{4}$ square it is within the quarter, and so on. As we generate our seeds by cycling the crumb pattern, the number of times a subpattern of some length shows up is exactly the number of seeds in the corresponding squarelet. If $N = K^4$ with some $K$ integer, then exactly one seed will fall in every cell of a $K^2 \times K^2$ grid (see Figure 7). This is very similar to the elemental interval property of the low discrepancy Halton sequence[8, 19].

Another obvious connection is that with *iterated function systems* (IFS)[2], and the *chaos game* method of generating their attractors[3]. Just like our construct, the chaos game transforms an initial point repeatedly. The transformation is randomly picked from a set each time. However, if the transformations map the attractor to disjunct areas, then it can be unambiguously determined for a point which the last transformation was. Then, starting with a point, the sequence can be traced back deterministically. In fact, the randomness is all encoded into the choice of the initial point. In our case, we are playing this deterministic version of the chaos game with four transformations, each mapping the unit square to one of its quarters. The attractor of this system is the unit square itself. We only take care that the sequence returns to itself, and thus we can work with a finite number of seeds. As for the crumb pattern, every crumb value there indicates one of the four transformations picked in the chaos game. If we use a uniform cycle, than all transformations appear the same number of times, and this is true for all sequences of transformations of given length, too.

Unfortunately, no polynomial-time algorithm is known for generating binary or quaternary uniform cycles of arbitrary length. In fact, we know of no proof that those exist for any $N$. However, in practice it is possible to find cycles of modest length by enumerating a set of required *snippets* and performing a brute force search over their permutations until a valid uniform cycle is found[17]. For $N = 16$, this can even be done manually, arranging the snippets 00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, yielding e.g. the quaternary sequence 0012202332131103. Note that all crumbs appear four times, and all snippets of two crumbs only once.

However, manual arranging for greater $N = K^4$ would be extremely ponderous, so it is required to define a suitable algorithm, based on the following considerations. Each snippet is a sequence of length $K$:

$$p_i = \{\pi_{i1}, ..., \pi_{iK}\}$$

Let $G = (P, E)$ be a directed graph, where $P = \{p_1, ..., p_N\}$ is the set of snippets, and $E = \{(p_i, p_j) \mid (\pi_{i2} = \pi_{j1}), ..., (\pi_{iK} = \pi_{j(K-1)})\}$. Thus, an edge connects two snippets if we get one from the other by dropping the first crumb and appending another one. A valid uniform cycle can be found by searching for a proper Hamiltonian cycle in the graph $G$. Starting with any vertex, the brute force method picks an edge randomly to an available position (see Figure 8), marking the current vertex as expended, and proceeding to the vertex along the selected edge. If there is no directed edge to an available vertex, the algorithm steps back to a previous state. Otherwise, we proceed similarly until a Hamiltonian cycle is found.

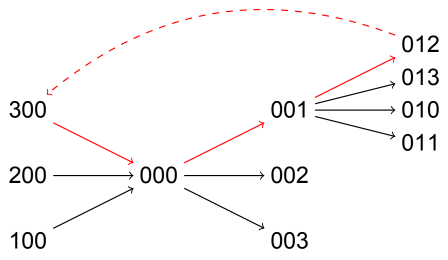

**Figure 8:** *A part of the graph for $N = 64$. We are looking for a Hamiltonian cycle.*

**Algorithm 1** Quaternary uniform cycle generation
```
1:  function UNIFORM(set of snippets $P$, sequence $Q$)
2:      if $|Q| \neq N$ then                              ▷ sequence incomplete
3:          $\rho \leftarrow$ random crumb from $(0,1,3,4)$
4:          for $\delta \leftarrow \rho, \rho+3 (\bmod 4)$ do     ▷ all continuations
5:              $p \leftarrow (q_{1-K}, q_{2-K}, ..., q_{-1}, \delta)$  ▷ form snippet
6:              if $p \in P$ then                         ▷ snippet available
7:                  $P' \leftarrow P \setminus p$                      ▷ expend snippet
8:                  $Q' \leftarrow Q \parallel \delta$                     ▷ append to $Q$
9:                  $\check{Q} \leftarrow$ UNIFORM$(P', Q')$              ▷ continue
10:                 if $\check{Q} \neq \emptyset$ then
11:                     return $\check{Q}$                           ▷ success
12:                 end if
13:             end if
14:             return $\emptyset$                 ▷ nothing worked, fail branch
15:         end for
16:     else                                          ▷ sequence complete
17:         for $i \leftarrow 0, K-1$ do           ▷ check wrapping snippets
18:             $p \leftarrow \left(q_{i-(K-1)}, q_{i-(K-2)}, \ldots, q_i\right)$
19:             if $p \notin P$ then
20:                 return $\emptyset$                     ▷ mismatch, reject $Q$
21:             end if
22:             $P \leftarrow P \setminus p$
23:         end for
24:         return $Q$                                      ▷ accept $Q$
25:     end if
26: end function
```

The formal algorithm (Algorithm 1) builds a growing sequence $Q = (q_0, q_1, \ldots, q_{-2}, q_{-1})$ of crumbs ultimately forming a quaternary uniform cycle. Note that the indices in $Q$ are understood $(\bmod |Q|)$, where $|Q|$ is the length of $Q$. The algorithm tests, for all possible continuations of $Q$,

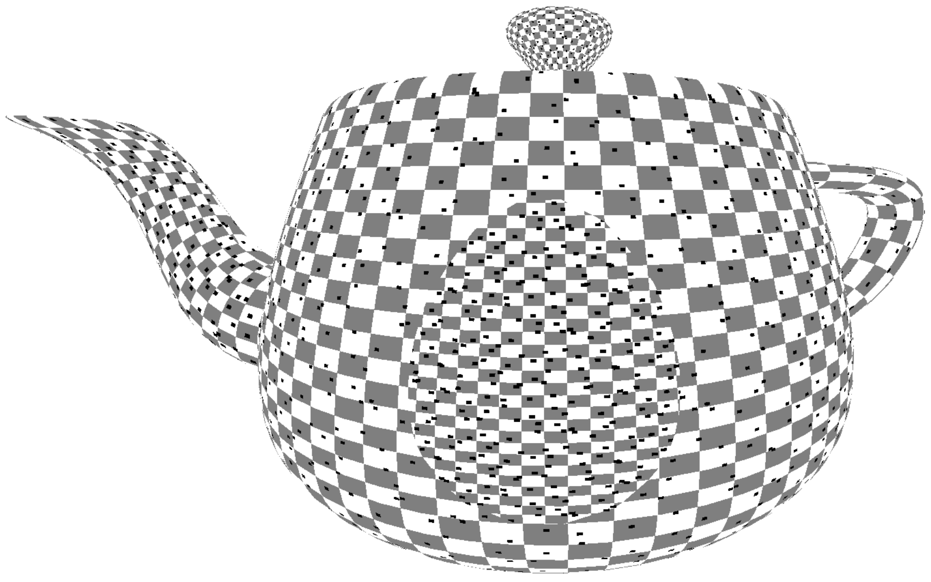

**Figure 7:** *Seeds generated by a uniform cycle are uniformly distributed in the sense that one seed falls in every grid cell.*

whether the resulting snippet is still available in set $P$. If it is, we expend the snippet from $P$ and append the continuation to $Q$. When $Q$ is complete, it is verified that the additional snippets generated by cycling the sequence are identical to those remaining in $P$. If they are, we have found a cycle containing all snippets once, and only once.

We can obtain a uniform cycle specifying the position for an initial seed by calling UNIFORM with an initial sequence of a random snippet (e.g. 000) and a snippet set $P$ with all possible snippets, save for the initial one in $Q$. Having obtained the crumbs of the initial seed, we get further seeds by repeatedly applying operator $\mathfrak{D}$ on these numbers, cycling the crumbs within the period.

## 5. Seed and stroke scale

Although, as evidenced by Algorithm 2 in Section 7, scaling seeds and strokes to match the required level of detail is fairly simple in practice, its rigorous discussion is quite involved. The process is analogous to that of *mipmapping*[22], the difference being that in our case all levels are identical, but scaled by powers of two. Thus some additional considerations for recursion and scaling between detail levels are needed.

As the scale of mapping UV texture coordinates to the viewport is changing, to preserve the same on-screen density, the hatching detail must be smoothly increased or decreased. Shall more detail be required, the original seed pattern should be repeated in all four quarters, continuing recursively until the desired seed density is achieved. To get lower density, the square should be regarded as a quarter of a bigger square, and only the relevant subset of the seeds used, also repeated recursively. Thus, from the mapping scale, a *nesting level* $\lfloor M \rfloor$ needs to be found, so that we know the scale at which the generated seeds can be interpreted as meaningful UV space positions.

Let $\mathcal{L}$ be the mapping of seeds to texture coordinates.

$$\mathbf{u} = \mathcal{L}\mathbf{s},$$

where $\mathbf{s}$ is a seed space position and $\mathbf{u}$ the texture coordinates. Locally, $\mathcal{L}$ must be an isotropic scaling. Globally, the scaling factor will vary for different points on the object surface, but it may change only by powers of two, to allow the self-similar nesting to work. For simplicity, we postulate that the scaling factor must always be a power of two. We show later in this section that this is without the loss of generality. Our objective is to find the scaling factor for any given surface point. In this discussion, we are looking for an arbitrary scaling factor first, and then select a power of two that approximates it.

Let $\mathcal{T}$ be the inverse of the texture mapping operator, defined by the model parametrization. Thus,

$$\mathbf{x} = \mathcal{T}\mathbf{u},$$

where $\mathbf{x}$ is the model space position.

Let $\mathcal{G}$ be the complete model-to-viewport-space mapping of the image synthesis pipeline, including the model, view, projection, and viewport transformations, all defined by object and camera setup. Thus,

$$\mathbf{v} = \mathcal{G}\mathbf{x},$$

where $\mathbf{v}$ is the viewport space position.

With these, the transformation from seed space to the viewport can be written as

$$\mathbf{v} = \mathcal{G}\mathcal{T}\mathcal{L}\mathbf{s}. \tag{1}$$

Note that all operations are dependent on the surface point, and some may be non-linear.

Let $\mathbf{h}$ be the *detail direction*, a differential direction vector in the seed space. If strokes are isotropic, e.g. only dots, then the choice is arbitrary. Typically, however, strokes have a dominant direction. While scaling in this dominant direction only influences stroke length, scaling perpendicularly has significant impact on hatching density. The detail vector should align with this perpendicular direction.

What we are interested in is how the detail direction is scaled by the mappings $\mathcal{G}$, $\mathcal{T}$, and $\mathcal{L}$. Therefore, let us introduce the notation

$$\text{stretch}(\mathcal{O}, \mathbf{d}) = \frac{|J_{\mathcal{O}}\mathbf{d}|}{|\mathbf{d}|},$$

where $\mathcal{O}$ is a mapping, $\mathbf{d}$ is a direction vector, and $J_{\mathcal{O}}$ the Jacobian matrix of mapping $\mathcal{O}$ at the surface point in question. Then the scaling factors exercised on direction $\mathbf{h}$ by the mappings can be written as:

$$L = \text{stretch}(\mathcal{L}, \mathbf{h}),$$

$$T = \text{stretch}(\mathcal{T}, J_{\mathcal{L}}\mathbf{h}),$$

$$G = \text{stretch}(\mathcal{G}, J_{\mathcal{T}}J_{\mathcal{L}}\mathbf{h}).$$

With these, Equation 1 can be linearized and applied to differentials, yielding the formula for the scaling of the detail direction as

$$|\mathbf{h}_{\text{vp}}| = GTL|\mathbf{h}|,$$

where $|\mathbf{h}_{\text{vp}}|$ is the length of the detail direction vector as it appears in the viewport. The geometry factor $G$ and texture distortion factor $T$ can be computed easily, and $L$ is the scaling that should be introduced by the choice of the detail level.

The ratio $F = |\mathbf{h}|/|\mathbf{h}_{\text{vp}}|$ captures how densely seeds appear in the viewport. This is a free artistic parameter, and a global constant, as we do not consider density modulation for tone yet. As with regular texture mapping, choosing a lower value of $F$ results in more detail—more seeds, thus more hatching strokes per unit area—, but also more repetition as the texture coordinates wrap around. With this, the desired detail factor $L^{-1}$ can be expressed as

$$L^{-1} = GTF,$$

where all factors are known. Note that if $L$ contained an additional constant scaling factor, it would have the same effect as $F$ here, so introducing another free parameter would be superfluous. Our postulation that $\mathcal{L}$ is a scaling with a power of two was without the loss of generality.

To make use of the nesting property, $\mathcal{L}$ should be a scaling with a power of two, thus $L \approx 2^{\lfloor M \rfloor}$, where we call integer $\lfloor M \rfloor$ the *nesting level*. We first compute the real number $M$ form $L^{-1}$ as

$$M = \log_2 L = -\log_2 L^{-1} = -\log_2 GTF,$$

then take the integer part to get the nesting level $\lfloor M \rfloor$. This gives us the scaling factor between seed space and texture coordinates as

$$\mathbf{u} = 2^{\lfloor M \rfloor}\mathbf{s}.$$

In practice, it is the texture coordinates of a surface point that are known when shading is performed, and the seed space coordinates need to be computed as

$$\mathbf{s} = 2^{-\lfloor M \rfloor}\mathbf{u}.$$

The solution is exact if $M$ is an integer, and integer values define *detail levels*. For non-integer values of $M$, the *dense level* $\lfloor M \rfloor$ has to transition into the *sparse level* $\lfloor M \rfloor + 1$ smoothly. Therefore, we need to scale strokes appropriately and fade out those that are not visible in the sparse level (Figures 9 and 10).

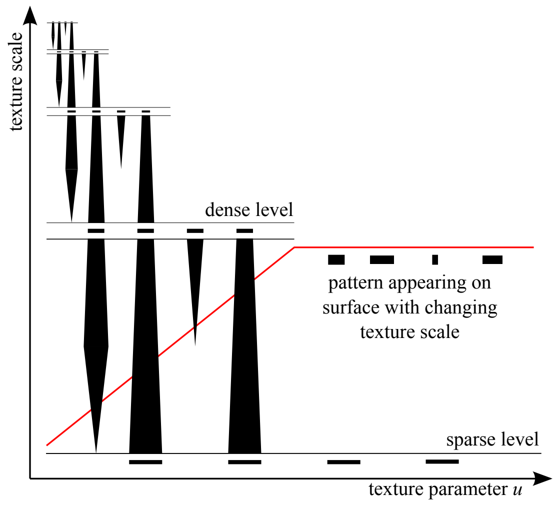

**Figure 9:** *2D depiction of smooth transition between detail levels by stroke width modulation.*

The stroke width and length in viewport space are artistic parameters, expressed as the two-dimensional vector $\mathbf{e}$. By the definition of $F$, we know that the seed space stroke size should be $F\mathbf{e}$, if $L$ were not quantized to powers of two. In

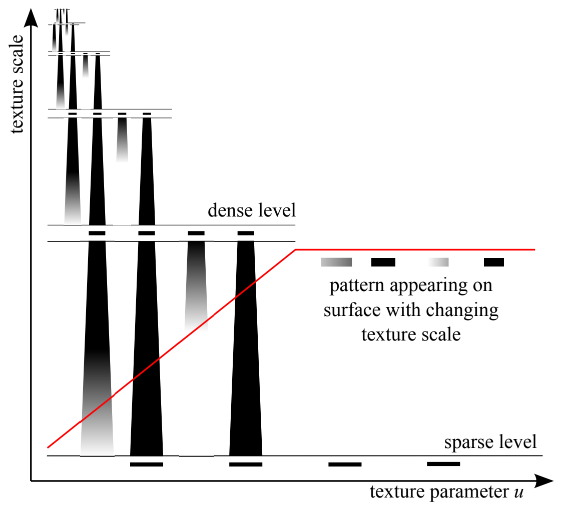

**Figure 10:** *2D depiction of smooth transition between detail levels by stroke opacity modulation.*

order to compensate for the quantization, we need to scale seed space stroke sizes by

$$\frac{2^M}{2^{\lfloor M \rfloor}} = 2^{M-\lfloor M \rfloor} = 2^{\mathrm{frac}(M)} = 2^m,$$

where $m$ can be seen as an interpolation factor between nesting levels, going from 0 at the dense level to 1 at the sparse level. Intuitively, if the dense and sparse levels are identical but for a factor of two, the strokes need to grow to twice their size as the nesting level increases.

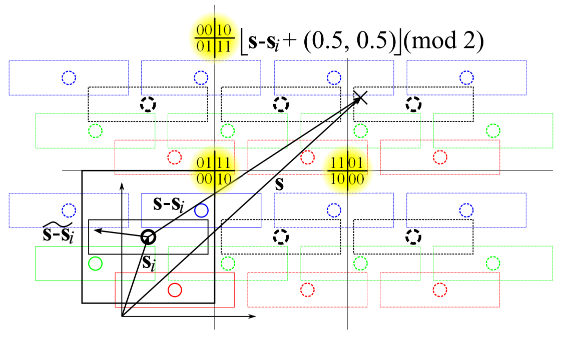
00 10
01 11
$\lfloor$s-s$_i$+ (0.5, 0.5)$\rfloor$(mod 2)
01 11
00 10
11 01
10 00
s
s-s$_i$
$\widetilde{\text{s-s}_i}$
s$_i$

**Figure 11:** *Computation of stroke space position (without scaling or rotation) and sparse level quarter indicator bits. X marks the shaded point. Its seed space position is* $\mathbf{s}$*, the seed processed is* $\mathbf{s}_i$*, the position relative to the seed is* $\mathbf{s}-\mathbf{s}_i$*, which is mapped to the unit square surrounding the seed to get* $\widetilde{\mathbf{s}-\mathbf{s}_i}$*.*

For every seed $\mathbf{s}_i$, we need to find *stroke space coordinates* $\mathbf{z}_i$ corresponding to seed space position $\mathbf{s}$. To cover the complete seed space, the seed pattern is repeated indefinitely in unit tiles. Strokes extend beyond tile boundaries (see Figure 11). Stroke extents in seed space must to be less than one, so that they do not overlap with themselves. The position of shaded point $\mathbf{s}$ relative to seed position is $\mathbf{s}-\mathbf{s}_i$, but this contains the offset of the tile. We introduce the following notation

$$\widetilde{(s_u, s_v)} = (\mathrm{frac}\,(s_u + 0.5) - 0.5, \mathrm{frac}\,(s_v + 0.5) - 0.5)$$

for wrapping the space to the origin-centered unit square. With this, the stroke coordinates are

$$\mathbf{z}_i = \mathbf{R}\widetilde{\mathbf{s}-\mathbf{s}_i} \oslash (2^m F\mathbf{e}),$$

with $\oslash$ standing for the elementwise division, and $\mathbf{R}$ is a rotation matrix in two dimensions for cross-hatching stroke alignment. Stroke coordinates can be used to access the stroke texture. Contributions for all seeds must be composited.

To be able to tell whether a dense seed is present on the sparse level, we need to know which quarter of the sparse level we are in, and whether the seed appears in that quarter. The parity bits of the tile's row and column indices indicate the quarter. Which tile we are is thus exactly identified by the integer part discarded with $\widetilde{\mathbf{s}-\mathbf{s}_i}$. This can be computed as

$$\mathbf{w} = \lfloor \mathbf{s} - \mathbf{s}_i + (0.5, 0.5) \rfloor.$$

Recall that the sparse level has the same seeds as the dense level, scaled up by a factor of two. A dense level seed that also appears in the sparse level must therefore be a scaled-up image of a seed in the dense level. This is true for every dense level seed that was generated by the $\mathfrak{D}$ operator from a seed in the respective quarter. To see what the predecessor of a seed was, we need to check the final bits in the cycled bit pattern. If those are identical to the parity bits of $\mathbf{w}$, then the seeds exits on both levels. Others have to be faded out as $m$ increases.

Fading strokes out can be accomplished in several ways. We can use alpha blending or stroke width modulation, and we can fade out all strokes simultaneously or one after the other, as the detail decreases. Modulating all strokes simultaneously allows them to blend smoothly, without abruptly appearing or disappearing strokes, but a large number of strokes will be semi-transparent or intermediate-sized, not achieving uniform hatching. If strokes appear one after the other, the method of modulation hardly matters, as they appear more abruptly, but the image space consistency is better. Note that because of the self similar property, hatching strokes only appear or disappear only when hatching needs to grow denser or sparser, and no flickering is present.

## 6. Tone

In order to convey illumination, we need to be able to modulate hatching density depending on the locally desired shade or tone. The challenge is to preserve the quality of hatching. In our scheme, seeds cannot be removed without breaking the recursive nesting property. The number of seeds could be decreased by decreasing the length of the quaternary uniform cycle, with the overall pattern remaining consistent and most seeds changing positions only slightly, but there would inevitably be seeds that have very different positions. Thus, the only option remains to use several sets of self-similar seeds, and overlay them. Fortunately, this is in perfect agreement with artistic practice[9], where about four distinct tones are rendered by overlaying strokes, usually at an angle, which is known as *cross-hatching*.

Thus, four seed sets are generated, and the contributions of a seed set are added if the desired local tone is darker than an associated threshold. However, to avoid clipping strokes at tone segment boundaries, this transition also has to be smooth. Again, strokes can be faded out together, producing a smoother animation, but with a lot of semi-transparent strokes, or one after the other, producing more abruptly appearing strokes of more consistent appearance.

## 7. Implementation

Once proper seed sets have been generated, the algorithm can be implemented in a single shader (Algorithm 2). The quaternary uniform cycles $\{b_1,\ldots,b_4\}$ defining the seed sets, rotation matrices for cross-hatching alignment $\{\mathbf{R}_1,\ldots,\mathbf{R}_4\}$, the global, non-modulated seed density $F$, and stroke size $\mathbf{e}$ are uniform global inputs. The smoothstep function $\mathrm{sstep}_{[\nu,\mu]}(\lambda)$ clamps and normalizes $\lambda$ to $[\nu,\mu]$, then performs a Hermite interpolation. The shader psuedocode presented here uses four hatching layers for tone. It fades strokes one after the other, using opacity modulation, both for tone and detail interpolation. The contributions of strokes are composited with the alpha blending logic, but within the shader. The stroke texture sampler must return zero alpha for out-of-range texture coordinates.

It has to be noted that this implementation uses $4N$ texture samples. With a seed set of 64 elements, this is 256 samples per pixel. Although these are samples from the same, presumably small stroke texture, this is still a brute force approach that can be improved if we filter strokes by proximity.

**Algorithm 2** Shading a surface point

```
function SHADE(texture coords $\mathbf{u}$, position $\mathbf{x}$)
    $\{b_1,\ldots,b_4\} \leftarrow$ uniform crumb cycles
    $F \leftarrow$ global seed density
    $\mathbf{e} \leftarrow$ stroke size
    $\{\mathbf{R}_1,\ldots,\mathbf{R}_4\} \leftarrow$ cross-hatching rotations
    $a \leftarrow$ tone from illumination in $[0,5]$
    $\mathbf{c} \leftarrow \mathbf{1}$                                    ▷ paper color
    for $j \leftarrow 1,4$ do                                ▷ for all tone layers
        $T \leftarrow$ texture distortion at $\mathbf{x}$
        $G \leftarrow$ geometry factor at $\mathbf{x}$
        $M \leftarrow -\log_2 GTF$                             ▷ detail factor
        $\mathbf{s} \leftarrow 2^{-\lfloor M \rfloor}\mathbf{u}$                                ▷ seed space
        $m \leftarrow M - \lfloor M \rfloor$
        for $i \leftarrow 0, N-1$ do                              ▷ for all seeds
            $\mathbf{s}_i \leftarrow 0. \parallel \mathbf{b}_j \parallel \mathbf{b}_j \parallel \ldots$       ▷ seed from crumbs
            $\alpha \leftarrow \mathrm{sstep}_{[\frac{i}{N+1},\frac{i+1}{N+1}]}(a-j)$            ▷ tone fade
            $\mathbf{w} \leftarrow \lfloor \mathbf{s}-\mathbf{s}_i+(0.5,0.5) \rfloor$       ▷ sparse quarter
            if $\mathbf{w} \not\equiv \mathbf{b}_j \pmod 2$ then           ▷ not in sparse
                $\alpha \leftarrow \alpha\,\mathrm{sstep}_{[\frac{i}{N+1},\frac{i+1}{N+1}]}(m)$      ▷ detail fade
            end if
            $\mathbf{z}_i \leftarrow \mathbf{R}\widetilde{\mathbf{s}-\mathbf{s}_i} \oslash (2^m F\mathbf{e})$          ▷ stroke space
            $\mathbf{y} \leftarrow \mathrm{strokeTex}[\mathbf{z}_i]$      ▷ sample from texture
            $\alpha \leftarrow \alpha y_\alpha$                           ▷ texture alpha
            $\mathbf{c} \leftarrow (1-\alpha)\mathbf{c}+\alpha\mathbf{y}$           ▷ alpha blending
            $\mathbf{b}_j \leftarrow \mathbf{b}_j \circlearrowleft 1$                      ▷ circular shift
        end for
    end for
    return c
end function
```

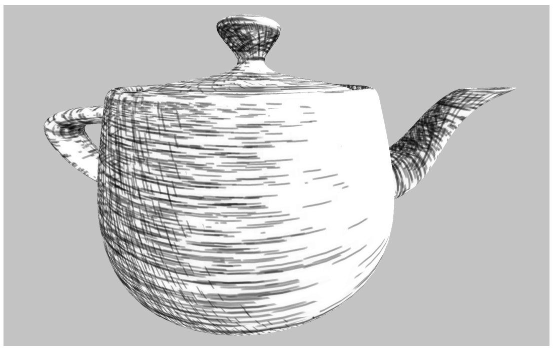

**Figure 12:** *Teapot rendered with* 4 *tone layers,* 64 *seeds per layer, at 20 FPS,* $1920 \times 1200$.

## 8. Results

We have tested the algorithm on an NVIDIA GeForce GTX 780, with $1920 \times 1200$ full-screen resolution. With four tone layers and 64 seeds per layer (Figure 12), we achieved an interactive performance of 20 frames per second. With 16 seeds (Figure 13), the performance went up to 80 FPS, confirming our expectation that rendering time is linearly proportional to the number of seeds.

We examined the options of fading in strokes between levels and tone layers simultaneously of one after the other. Simultaneous fading (Figure 14) always resulted in thin or semi-transparent strokes appearing in conspicuous patterns. Fading strokes individually (Figure 15) resulted in just the occasional stroke being in a transient state briefly, acceptably simulating the stroke appearing as a result of an artistic process. The overall image consistency was in this case perfect. With strokes fading in individually, whether we used

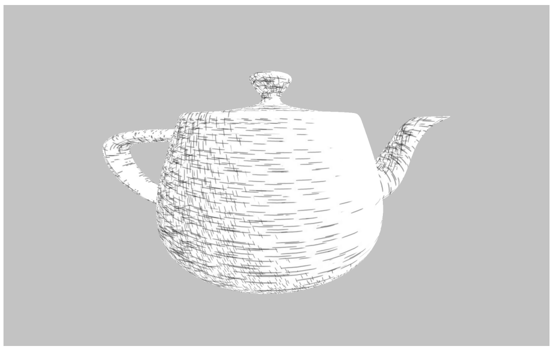

**Figure 13:** *Teapot rendered with* 4 *tone layers,* 16 *seeds per layer, at 80 FPS,* $1920 \times 1200$.

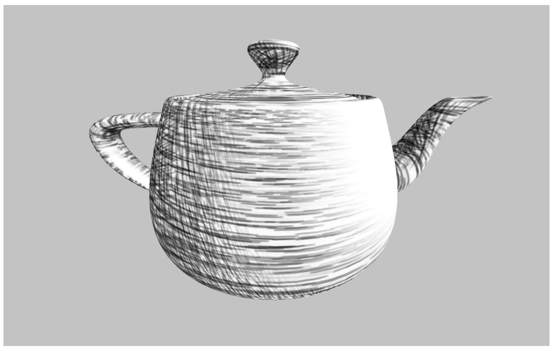

**Figure 14:** *Teapot rendered with simultaneous stroke fading.*

opacity or line width weighting for fading did not cause a significant visual difference. This was due to the fact that in any method based on local shading like ours, detail level can change strongly along a stroke. Some parts of the stroke will therefore be completely visible and others completely invisible, and the transition is both spatially and temporally confined.

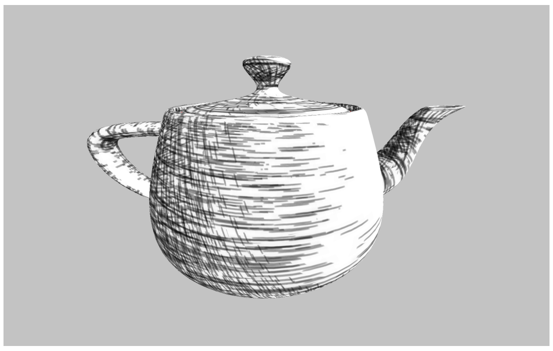

**Figure 15:** *Teapot rendered with individual stroke fading.*

In animation, the object space coherence, as expected with a parameter space approach, was impeccable. Flickering was not observed. The pattern in which the strokes appear and disappear when zooming in or out is conceivably random.

## 9. Future work

Our method could be significantly accelerated by not considering all seeds in every pixel. For this, we need to create a texture representing the unit square in seed space, where every texel contains a list of those strokes that may overlap with the texel. This can simply be pre-generated for any seed be rendering all strokes at their maximum size into an S-buffer[21]. We expect that with this addition the proposed method will be barely slower that simple texturing.

When a surface is visible at an integer detail factor, all strokes are completely visible. Otherwise, some may be in a transient state fading in. This difference is unnoticeable when strokes fade individually, but we conjecture that this is the reason why simultaneous fading does not produce unacceptable results. Therefore, we would like to introduce theory for mixed-weight detail levels, where integer level patterns are indistinguishable from interpolated ones. We also believe this will lead us to the implementation of a new artistic parameter that allows intermediate strategies between simultaneous and individual fading. In this overlapped fading model, a customizable, but fixed percentage of strokes will be in transitional state at any time and detail level.

We also plan to examine interaction with outline rendering approaches, and investigate whether we can simulate overdraw with screen space filtering.

## 10. Acknowledgements

This work has been supported by OTKA PD-104710 and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

## References

1. Zainab AlMeraj, Brian Wyvill, Tobias Isenberg, Amy A Gooch, and Richard Guy. Automatically mimicking unique hand-drawn pencil lines. *Computers & Graphics*, 33(4):496–508, 2009.
2. Michael F Barnsley and Stephen Demko. Iterated function systems and the global construction of fractals. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 399(1817):243–275, 1985.
3. Michael F Barnsley and Andrew Vince. The chaos game on a general iterated function system. *Ergodic Theory and Dynamical Systems*, 31(04):1073–1079, 2011.

4. Derek Cornish, Andrea Rowan, and David Luebke. View-dependent particles for interactive non-photorealistic rendering. In *Graphics interface*, volume 1, pages 151–158, 2001.
5. Oliver Deussen, Stefan Hiller, Cornelius Van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. In *Computer Graphics Forum*, volume 19, pages 41–50. Wiley Online Library, 2000.
6. Ahna Girshick, Victoria Interrante, Steven Haker, and Todd Lemoine. Line direction matters: an argument for the use of principal directions in 3d line drawings. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 43–52. ACM, 2000.
7. Paul Haeberli. Paint by numbers: Abstract image representations. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 207–214. ACM, 1990.
8. John H Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM*, 7(12):701–702, 1964.
9. A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.
10. Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by example: a statistical approach. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 29–36. ACM, 2002.
11. Yongjin Kim, Jingyi Yu, Xuan Yu, and Seungyong Lee. Line-art illustration of dynamic and specular surfaces. In *ACM Transactions on Graphics (TOG)*, volume 27, page 156. ACM, 2008.
12. Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 37–45. ACM, 2006.
13. Yunjin Lee, Lee Markosian, Seungyong Lee, and John F Hughes. Line drawings via abstracted shading. In *ACM Transactions on Graphics (TOG)*, volume 26, page 18. ACM, 2007.
14. Barbara J Meier. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484. ACM, 1996.
15. Afonso Paiva, Emilio Vital Brazil, Fabiano Petronetto, and Mario Costa Sousa. Fluid-based hatching for tone mapping in line illustrations. *The Visual Computer*, 25(5-7):519–527, 2009.
16. Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581. ACM, 2001.
17. D.J. Spitzner. Searchable randomness. Technical Report 09-01, Department of Statistics, University of Virginia, January 2009.
18. Thomas Strothotte and Stefan Schlechtweg. *Non-photorealistic computer graphics: modeling, rendering, and animation*. Elsevier, 2002.
19. L. Szirmay-Kalos. *Monte-Carlo Methods in Global Illumination — Photo-realistic Rendering with Randomization*. VDM, Verlag Dr. Müller, Saarbrücken, 2008.
20. Tamás Umenhoffer, László Szécsi, and László Szirmay-Kalos. Hatching for motion picture production. In *Computer Graphics Forum*, volume 30, pages 533–542. Wiley Online Library, 2011.
21. Andreas A Vasilakis and Ioannis Fudos. S-buffer: Sparsity-aware multi-fragment rendering. In *Eurographics 2012-Short Papers*, pages 101–104. The Eurographics Association, 2012.
22. Lance Williams. Pyramidal parametrics. In *ACM Siggraph Computer Graphics*, volume 17, pages 1–11. ACM, 1983.
23. Georges Winkenbach and David H Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 91–100. ACM, 1994.
24. Andrew P Witkin and Paul S Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277. ACM, 1994.
25. Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. High quality hatching. In *Computer Graphics Forum*, volume 23, pages 421–430. Wiley Online Library, 2004.