Recursive Procedural Tonal Art Maps

László Szécsi Marcell Szirányi
Budapest University of Technology and Economics
Magyar Tudósok körútja 2
H-1117, Budapest, Hungary
szecsi@iit.bme.hu sziranyi@iit.bme.hu

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 recursive 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.

Keywords

NPR, hatching, tonal art maps, self-similar

1 INTRODUCTION

Image synthesis methods mimicking artistic expression and illustration styles [Hae90, PHWF01, SS02] are usually vaguely classified as *non photo-realistic rendering* (NPR). 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. In particular, when surface distance or viewing angle is changing, object-space density of strokes should adapt without the strokes flickering or drifting on the surface, while presenting natural randomness inherent in manual work [JEGPO02, AWI*09].

In this paper we present *recursive procedural tonal art maps* (RPTAM), a single-shader rendering technique that fulfils the above criteria with less limitations than previous techniques (Figure 1). In particular, strokes are preserved as constructing elements (as opposed to texture harmonics), infinite zooming is possible (as opposed to a finite LOD set), and local shading is sufficient (as opposed to global geometry processing and hidden stroke removal). The key idea is that we place strokes in texture space, at pre-generated seed locations exhibiting a self-similar pattern, allowing for smooth transitions between any texture scalings.

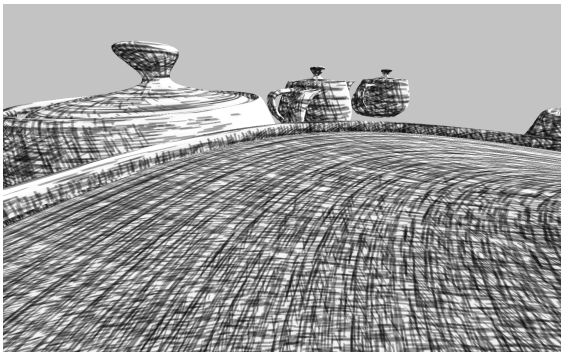

Figure 1: Teapots rendered with the same shader and settings, featuring different levels of detail.

The organization of the paper is as follows. In Section 2 we summarize the related previous work on both hatching and self-similarity. Section 3 introduces the idea of Recursive Procedural Tonal Art Maps. In Section 4, we derive the mathematical construct for the placement of hatching strokes to meet the self-similarity requirements. In Section 4.1, we propose a scheme to generate good quality seed sets. We discuss rendering in Section 5, including the level-of-detail scheme and tone representation. The 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 opacity of thin hatching strokes [WS94, HZ00]. To mimic this, we should find a *density* and a *direction field* in the image plane that is

 ISBN 978-80-86943-70-1

as close as possible to what an artist would use. The density, length, width and opacity should be influenced by the current illumination, while the orientation is determined by the underlying geometry. Artists may use *cross-hatching*, i.e. several layers of strokes, aligned at different angles to the direction field.

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

2.2 Seed-based hatching

Several works [Mei96, USSK11] 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 [ZISS04, PBPS09]. 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 [WH94]—even using mesh subdivision and simplification [CRL01]—, or rejection sampling [USSK11]. 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 hatching curves against triangle mesh objects must be using heavy bias and smooth rejection to avoid flickering.

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, but use their positions for procedural texturing.

2.3 Self-similarity in NPR

Dynamic solid textures [BBT09] have been proposed for infinite zooming capability in stylistic rendering. The textures are fractalized, and represented as a weighted sum of octaves, whose scale and blending weights change in a cyclic manner to produce zooming. Our work shares the same motivation, and we also exploit self-similarity, but we keep strokes as generating elements, focusing on hatching rather than generic texturing, thus avoiding contrast loss and mixing in unintended frequencies.

The Halton sequence has been used to generate seed sequences [USSK11] that can be truncated to get statistically similar, lower density hatching. As any number of seeds can be generated this way, this allows for infinite level of detail. We aim at the same effect, but working in texture space to avoid stroke geometry processing and hidden stroke removal complications, and with a finite, but self-similar seed set to eliminate the need of generating a view-dependent number of seeds.

2.4 Texture-based hatching

In order to assure that strokes remain fixed to 3D surfaces, hatches can be generated into textures and mapped onto animated objects [LKL06]. This requires surface parametrization. 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.

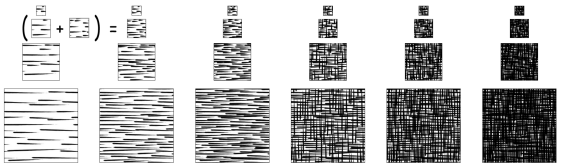

Figure 2: A Tonal Art Map with the nesting property. Strokes in one image appear in all the images to the right and down from it. From Praun et. al [PHWF01].

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 [PHWF01]. 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 different-detail 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 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 RECURSIVE TONAL ART MAPS

Our goal is to unify the simplicity and robustness of Tonal Art Maps with the unlimited level-of-detail offered by deterministic seed generation and rejection sampling. To achieve this, we make Tonal Art Maps infinitely loopable, by making the nesting property recursive.

The workflow of the complete solution is:

- Bit patterns—fitting in a few bytes—representing recursively nested seed sets are generated offline.
- A stroke coverage texture is rendered that stores seed IDs, indicating which strokes could contribute to a surface point. This texture must only be re-rendered when the hatching style (e.g. overall stroke width) is modified.
- When shading a surface point, the required detail level is computed. Scaled according to the detail level, the stroke coverage texture is queried for IDs of relevant seeds. Their positions are reconstructed from the bit pattern and the ID. Hatching strokes—using an artist-drawn stroke texture, scaled and faded for required detail—are placed at these positions and processed for their contribution to the color of the shaded point.

Strokes are positioned at *seeds* in the texture representing the hatching pattern. This texture is broken into four tiles, forming a 2×2 grid. In all four quarters, the seeds must be the subset of the complete seed set scaled down to fit the quarter. This way, the seeds are nested in the pattern that we get by repeating them twice along both axes (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. 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 positioning seeds, in a way that assures the recursive nesting property. In Section 4.1, we discuss what seed sets produce good quality hatching patterns and how to generate those. We address hatching stroke length, width, and orientation in Section 5.

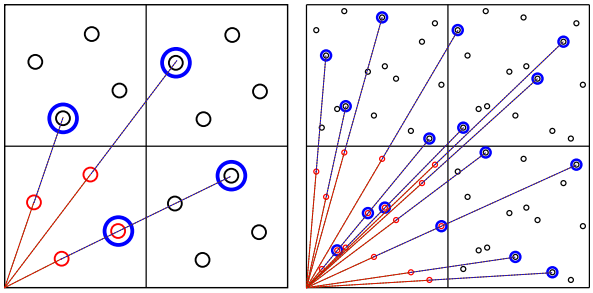

Figure 3: Seed sets of 4 and 16 elements with the recursive nesting property. Large circles indicate seeds, small circles are the seed pattern repeated on a 2×2 grid. The dense pattern scaled up from any corner gives the sparse pattern.

4 SEED GENERATION

Our construction of seed point positions unambiguously follows from the requirement of the recursive nesting property, and that we use a finite seed set.

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*. Later, to get UV space positions, seed space positions will be scaled according to the detail level, and wrapped over indefinitely.

In order to formalize zooming in to any quarter of the unit square, let us define the operator $\mathcal{D}$ as

$$\mathcal{D}\mathbf{s} = \langle 2\mathbf{s} \rangle,$$

with angle brackets denoting the fractional part. Geometrically, this operation is a scaling by the factor of two, using the nearest corner of the unit square as the center. This tells us where a seed gets when we zoom in to the quarter it is in. The recursive nesting property requires that all those positions coincide with seeds, as the zoomed-in version must be nested in the complete pattern.

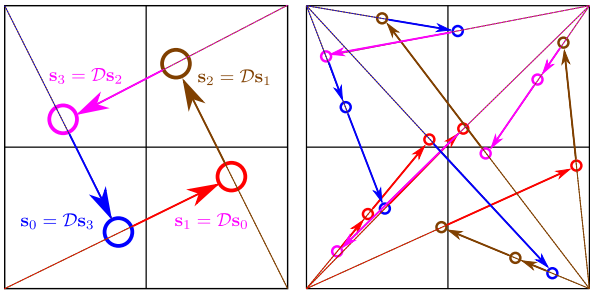

Figure 4: Operator $\mathcal{D}$ projects seeds to double their distance from the nearest corner. Seed sequences are generated by repeating this operation. Seeds are colored to show the corner used for the scaling.

 ISBN 978-80-86943-70-1

This means that if we already know a seed $\mathbf{s}_i$, then its image $\mathcal{D}\mathbf{s}_i$ must also be a seed (Figure 4). A sequence of N seeds then must be found as

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

However, as our set must be finite, $\mathcal{D}\mathbf{s}_{N-1}$ must also be in S. This can be true if $\mathbf{s}_0 = \mathcal{D}\mathbf{s}_{N-1}$.

There is an obvious connection to *iterated function systems* (IFS) [BD85], and the *chaos game* method of generating their attractors [BV11]. 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 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.

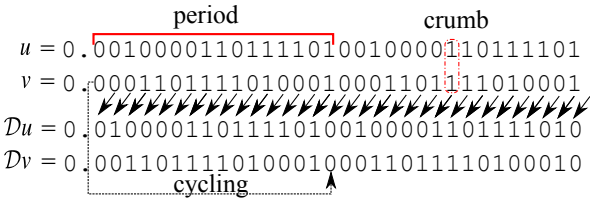

Figure 5: Interpretation of operator $\mathcal{D}$ on binary fractions. For periodic fractions, the bits can be cycled.

In order to find an initial seed that produces such a closed seed loop, let us consider the binary radix fraction form of the seed coordinates. The operator $\mathcal{D}$ removes the first binary digit after the binary radix point, shifting the rest to the left (Figure 5). Thus, we can interpret the above method of seed generation as repeatedly shifting the bits of coordinates s_{0u} and s_{0v}. 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). Any such periodic binary fractions produce a recursively nested seed set, but not all of them are distributed evenly and isotropically. Therefore, in Section 4.1 we propose a method for finding binary fractions that represent good quality seed sets.

4.1 Uniform seed sets

In an infinite sequence of independent, random elements with discrete, uniform outcome probabilities, we expect any fixed-length pattern to appear with the same frequency. Extending that to finite sequences, we expect possible fixed-length patterns to appear with frequencies as uniform as possible when cycling through the sequence. Bit sequences with such a property are called *uniform cycles* [Spi09]. The bit sequence for the u and v coordinates could be found independently, but then coincidentally identical patterns could appear, making u and v correlated, and the seed pattern anisotropic. Instead, we can combine the respective bits of u and v periods 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. When we generate seeds by cycling the bits, the number of times a pattern of some length shows up behind the radix point is exactly the number of seeds in the corresponding squarelet. If $N = 4^K$ with some K integer, then exactly one seed will fall in every cell of a $2^K \times 2^K$ grid (Figure 6). This is very similar to the elemental interval property of the low discrepancy Halton sequence [Hal64].

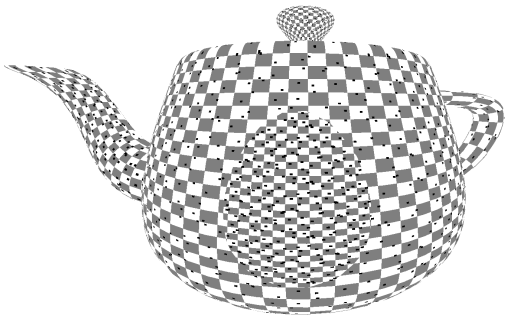

Figure 6: Seeds generated by a uniform cycle are uniformly distributed in the sense that one seed falls in every grid cell.

Although no polynomial-time algorithm of generation is known for binary or quaternary uniform cycles of arbitrary length, in practice it is possible to find cycles by enumerating a set of all possible *snippets* of length $\log_4 N$ and performing a brute force search over their permutations [Spi09]. A permutation is valid if all snippets, with their first crumb removed, are identical to the consecutive one without its last crumb. The uniform cycle is given by the initial crumbs of all snippets. 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 uniformity applies: all

crumbs appear four times, and all possible snippets of two crumbs only once.

Algorithm 1 Quaternary uniform cycle generation

```
1:  function UNIFORM(set of snippets P, sequence Q)
2:      if |Q| ≠ N then                          ▷ sequence incomplete
3:          ρ ← random crumb from (0,1,2,3)
4:          for δ ← ρ, ρ+3 (mod 4) do            ▷ for all 4
5:              p ← (q_{1−K}, q_{2−K}, ..., q_{−1}, δ)   ▷ postfix
6:              if p ∈ P then                    ▷ pfx snippet available
7:                  P′ ← P \ p                   ▷ expend snippet
8:                  Q′ ← Qδ                      ▷ append crumb to Q
9:                  Q̌ ← UNIFORM(P′, Q′)          ▷ continue
10:                 if Q̌ ≠ ∅ then
11:                     return Q̌                 ▷ success
12:         return ∅                             ▷ nothing worked, fail branch
13:     else                                     ▷ sequence complete
14:         for i ← 0, K−1 do                    ▷ check wrapping
15:             p ← (q_{i−(K−1)}, q_{i−(K−2)}, ..., q_i)
16:             if p ∉ P then                    ▷ wrapped snippet
17:                 return ∅                     ▷ mismatch, reject Q
18:             P ← P \ p                        ▷ expend wrapped snippet
19:         return Q                             ▷ accept Q
```

For an arbitrary N, we provide here an algorithm (Algorithm 1) that 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 four crumb values, whether appending them to Q results in a new postfix that 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 that specifies the position for an initial seed by calling UNIFORM with an initial sequence of a random snippet (e.g. 000) as Q, and a snippet set P with all possible snippets, save for the initial one already in Q. Having obtained the crumbs of the initial seed, we get further seeds by repeatedly applying operator $\mathcal{D}$ on these numbers, cycling the crumbs within the period. Note that Algorithm 1 only needs to be run once to get the bit pattern for a seed set, which can then even be hardwired into an implementation. In runtime, obtaining an individual seed from the bit pattern only needs a circular bit-shift operation.

5 RENDERING

We need to wrap the seed pattern on surfaces at an appropriate scale to ensure the desired viewport-space stroke size and density. This scale varies over the surface, and the pattern needs to adapt continuously in both time and space. We achieve this by first scaling the seed pattern by an approximating power of two (Section 5.1), then scaling the strokes themselves to make up for the difference (Section 5.2), and fading some strokes to transition between densities (Section 5.3).

5.1 Seed pattern scale

The process of detail level selection is analogous to that of *mipmapping* [Wil83], the difference being that in our case all detail levels are identical, but scaled by powers of two. In mipmapping, level selection is based on the viewport-space derivatives of UV coordinates. This is also true for our method, with two main differences. First, hatching patterns are anisotropic. If perspective distortion causes strokes merely to appear shortened, that bears very little stylistic impact, and strokes should not be made wider and sparser. If perspective distortion causes strokes to appear thinner and denser, however, a change in level of detail is justified. Thus, instead of taking the maximum of the rates of change along the u and v directions as in classic mipmapping, we are interested in the rate of change perpendicular to the hatching strokes. Second, the choice of the detail level does not indicate a choice from a precomputed set of textures, but it tells us at what scale the seeds should be wrapped over the UV space. As the overall goal is to achieve a given viewport-space density, this mapping between seed and UV spaces should cancel out the mapping between UV and viewport spaces as much as possible. Note that these mappings might not be linear or even well-defined globally, but they are always interpretable for a given surface point locally.

Let $\mathbf{L}$ be the unknown, local mapping of seeds to texture coordinates we wish to find:

$$\mathbf{x}_{\mathrm{uv}} = \mathbf{L}\mathbf{x}_{\mathrm{s}}$$

where $\mathbf{x}_{\mathrm{s}}$ is a seed space position $\mathbf{x}_{\mathrm{uv}}$ the texture coordinates, and $\mathbf{L}$ must be an isotropic scaling by a power of two. 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 $\mathbf{T}$ be the local inverse of the texture mapping operator, defined by the model parametrization. Thus,

$$\mathbf{x}_{\mathrm{obj}} = \mathbf{T}\mathbf{x}_{\mathrm{uv}}$$

where $\mathbf{x}_{\mathrm{obj}}$ is the model space position. This is the well-known tangent space to object space transformation, with the tangent and binormal vectors being the non-normalized partial derivatives of object space position with respect to u and v.

Let $\mathbf{G}$ 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,

$$\mathbf{x}_{\text{vp}} = \mathbf{G}\mathbf{x}_{\text{obj}}$$

where $\mathbf{x}_{\text{vp}}$ is the viewport space position.

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

$$\mathbf{x}_{\text{vp}} = \mathbf{GTL}\mathbf{x}_{\text{s}}. \quad (1)$$

Note that all operations are dependent on the surface point, and they are all local linearizations.

Let $\mathbf{h}$ be the *detail direction*, a differential direction vector in the seed space that is perpendicular to stroke direction. What we are interested in is how the detail direction is scaled by the mappings $\mathbf{G}$, $\mathbf{T}$, and $\mathbf{L}$. Let us define these scaling factors as the detail factor L, the texture distortion T, and the geometry factor G:

$$L = \frac{|\mathbf{Lh}|}{|\mathbf{h}|}, \quad T = \frac{|\mathbf{TLh}|}{|\mathbf{Lh}|}, \quad G = \frac{|\mathbf{GTLh}|}{|\mathbf{TLh}|}.$$

With these, Equation 1 can be 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 (we skip the formulae as both model-view-projection and tangent space transformations are well-known), 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 can be expressed as

$$L = (GTF)^{-1}.$$

To make use of the nesting property, 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 depth*. We first compute the real number M as

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

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

$$\mathbf{x}_{\text{s}} = 2^{-\lfloor M \rfloor}\mathbf{x}_{\text{uv}}.$$

The solution is exact if M is an integer, and integer values define *nesting 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 7 and 8).

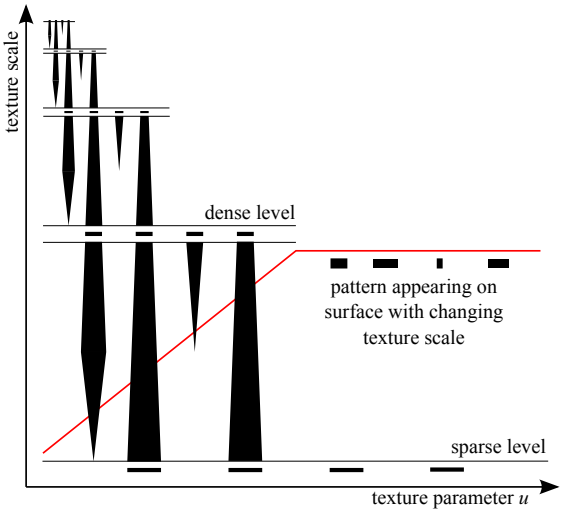

Figure 7: 2D depiction of smooth transition between nesting levels by stroke width modulation.

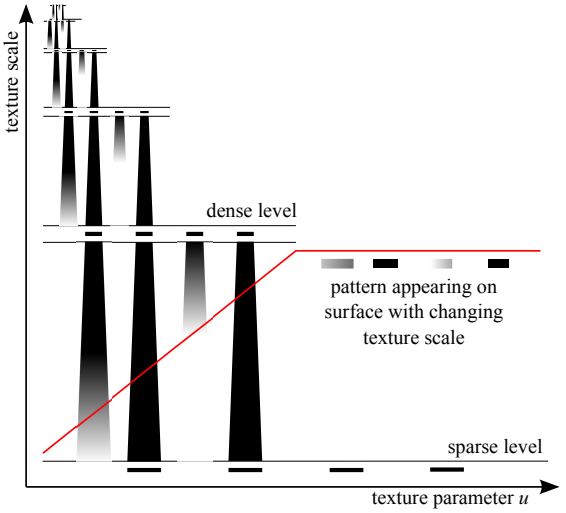

Figure 8: 2D depiction of smooth transition between nesting levels by stroke opacity modulation.

5.2 Stroke scale interpolation

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

$$\frac{2^M}{2^{\lfloor M \rfloor}} = 2^{M-\lfloor M \rfloor} = 2^{\langle M \rangle} = 2^m,$$

 ISBN 978-80-86943-70-1

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. Knowing that the dense and sparse levels are identical but for a factor of two, it is easy to picture why the strokes need to grow to twice their size as the interpolation factor increases.

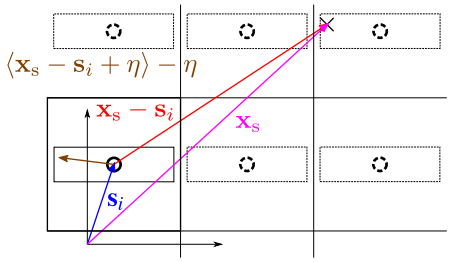

Figure 9: Computation of stroke space position (without scaling or rotation). $\times$ marks the shaded point. Its seed space position is $\mathbf{x}_s$, the seed processed is $\mathbf{s}_i$, the position relative to the seed is $\mathbf{x}_s - \mathbf{s}_i$, which is mapped to the unit square surrounding the seed as $\langle \mathbf{x}_s - \mathbf{s}_i + \eta \rangle - \eta$.

In order to find the color contribution of strokes at $\mathbf{x}_s$, we need to find which strokes cover this point, and what part of their stroke texture appears there. Thus, for every seed $\mathbf{s}_i$, we need to find *stroke space coordinates* $\mathbf{z}_i$ corresponding to seed space position $\mathbf{x}_s$ (Figure 9). The position of shaded point $\mathbf{x}_s$ relative to the seed position is $\mathbf{x}_s - \mathbf{s}_i$, but seeds must be wrapped around indefinitely to cover the entire plane. Thus, $\mathbf{x}_s - \mathbf{s}_i$ contains an integer offset corresponding to the unit tile in which the seed instance is found. We would like strokes to be centered at seeds, so we remove the integer offset by mapping $\mathbf{x}_s - \mathbf{s}_i$ to the origin-centered unit tile as $\langle \mathbf{x}_s - \mathbf{s}_i + \eta \rangle - \eta$, where $\eta = (1/2, 1/2)$. This gives us the relative position of point $\mathbf{x}_s$ to the nearest instance of seed $\mathbf{s}_i$. This can be rotated for stroke direction and scaled for stroke length and width, yielding the formula for the stroke texture coordinates

$$\mathbf{z}_i = (\mathbf{R}(\langle \mathbf{x}_s - \mathbf{s}_i + \eta \rangle - \eta)) \oslash (2^m F \mathbf{e}) + \eta,$$

where $\mathbf{R}$ is a rotation matrix for stroke alignment, with $\oslash$ standing for the elementwise division. These coordinates can be used to access the stroke texture. Contributions for all seeds must be composited.

5.3 Density interpolation

When we zoom out from a surface, as object-space seed density needs to decrease, we see the stroke pattern of the dense level fade to the sparse level. Some strokes grow in seed-space size in order to maintain their extents in viewport space (as described in Section 5), others vanish to make the hatching sparser. When interpolating between two levels, we need to identify those seeds that remain.

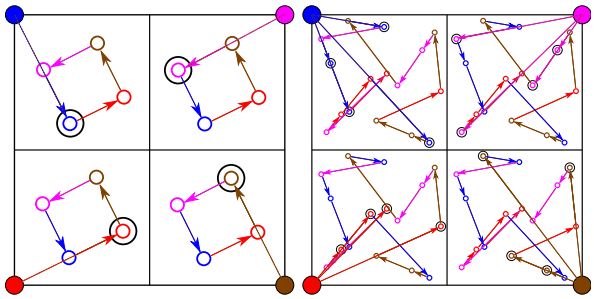

Figure 10: In the blue dense tile, blue seeds coincide with sparse seeds. A blue arrow in the blue tile is both a step in the dense tile's chaos game and the mapping of a dense seed to a sparse seed.

Four dense tiles cover a single sparse tile (Figure 10). Let us consider the dense seeds in the upper left (blue) tile. During the chaos game process, every one of those seeds has been constructed as a scaled image of another dense seed, some of them—the blue seeds—using the upper left corner as the center. Both blue seeds and sparse seeds are thus defined as dense seeds scaled out from the blue corner—therefore, they coincide. The same line of though can be followed for all tiles and corners. Thus, whether a dense seed coincides with a sparse seed depends on the agreement of two indicators: which of the four dense tiles it is in, and which of the four possible scalings produced it in the chaos game.

The parity bits of a dense tile's row and column indices indicate which tile it is, within the sparse level tile. The tile indices are exactly the integer part that was discarded with $\langle \mathbf{x}_s - \mathbf{s}_i + \eta \rangle$. This can be computed as

$$\mathbf{w} = \lfloor \mathbf{x}_s - \mathbf{s}_i + \eta \rfloor.$$

Which scaling is applied in the chaos game depends on the first bits of s_{iu} and s_{iv}. With cycling the bits, these become the final bits in the bit pattern of the new seed. Thus, a dense seed coincides with a sparse seed **iff** the parities of its u and v bit cycles are the same as that of the tile row and column indices $\mathbf{w}$. When zooming out, these seeds remain, while other seeds have to be faded out as interpolation factor m increases.

Fading strokes out can be accomplished in several ways. We can use opacity or stroke size modulation, and we can fade out all strokes simultaneously or one after the other, as the interpolation factor increases. 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 they all look similar. Note that because of the recursive nesting property, hatching

strokes appear or disappear only when hatching needs to grow denser or sparser, and no flickering is present.

5.4 Tone

In order to convey illumination, we need to be able to modulate hatching density depending on the locally desired shade or tone. Just as seeds can be faded out for less detail, their size or opacity can also be modulated by tone. 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.

Furthermore, in artistic practice [HZ00], tone is often emphasized by rendering about four distinct layers of strokes, usually at an angle, which is known as *cross-hatching*. To simulate that, we use several sets of self-similar seeds, and overlay them. All layers are associated with a tone range, and strokes within a layer are modulated when local tone is within that range. Thus, we avoid sharply clipping strokes at tone segment boundaries.

6 IMPLEMENTATION

Once a proper seed set, in the forms of u and v bit patterns, has been pre-generated, the algorithm can be implemented in a single shader (Algorithm 2). For didactic reasons and ease of understanding, the pseudocode listing was compiled with the following simplifications:

We do not use multi-layer hatching, to which the algorithm can easily be extended by mapping the desired tone to the tone ranges associated with the layers, and executing the algorithm for all layers, with possibly different bit patterns, stroke sizes, rotations, and stroke textures.

We use simultaneous modulation, as opposed to fading one stroke after the other, for both tone and detail. If we wish to fade strokes individually, we need to apply a smoothstep function on tone a and/or interpolation factor m, over the interval $[i/(N+1),(i+1)/(N+1)]$, in lines 11 and/or 14.

We use opacity modulation, but modulation of stroke size instead is straightforward.

We process all seeds of the set in a brute force iteration. In practice, only a few strokes influence the color of a surface point, so the loop of line 9 can be replaced by a much more economic solution detailed in Section 6.1.

Algorithm 2 Shading a surface point. Global inputs are the uniform bit cycles $\mathbf{b} = (b_u, b_v)$ defining the seed set, and the following artistic parameters: the rotation matrix for stroke alignment $\mathbf{R}$, the global, non-modulated viewport seed density F, and stroke width and height $\mathbf{e}$. The artist-drawn stroke texture strokeTex returns zero alpha for out-of-range texture coordinates.

```
1:  function SHADE(texture coords x_uv, position x_obj)
2:      a ← tone from illumination in [0,1]
3:      c ← 1                          ▷ init pixel color to paper color
4:      T ← texture distortion at x_obj
5:      G ← geometry factor at x_obj
6:      M ← −log2 GTF                                  ▷ nesting depth
7:      x_s ← 2^−⌊M⌋ x_uv                              ▷ UV to seed space
8:      m ← M − ⌊M⌋                                    ▷ interpolation factor
9:      for i ← 0, N−1 do                              ▷ for all seeds
10:         s_i ← (0.b_u b_u ..., 0.b_v b_v ...)       ▷ bits to seed
11:         α ← a                                      ▷ opacity from tone fade
12:         w ← ⌊x_s − s_i + η⌋                        ▷ dense tile index
13:         if w ≢ b (mod 2) then                      ▷ seed not in sparse
14:             α ← α(1 − m)                           ▷ detail fade
15:         z_i ← (R(⟨x_s − s_i + η⟩ − η)) ⌀ (2^m F e) + η
16:         y ← strokeTex[z_i]                         ▷ sample texture
17:         α ← α y_α                                  ▷ apply texture alpha
18:         c ← (1 − α)c + α y                         ▷ alpha blending
19:         b ← b ↻ 1                                  ▷ cycle bits for next seed
20:     return c
```

6.1 Seed pre-filtering

The brute force implementation uses N texture samples for every hatching layer. With 4 layers and a seed set of 64 elements, this would be 256 samples per pixel. In spite of this, a naive implementation does not perform as poorly as one could expect. The 256 samples are from the same, presumably small stroke texture, efficiently cached, with most texture fetches resolved without memory access for being out of bounds, and processing overhead is masked by the latency of the memory reads that do happen. Even so, the full potential of the approach can be achieved if we only process those strokes that potentially contribute to the shading of a surface point.

In order to identify these we create a *stroke coverage texture* representing the unit square in seed space, where every texel contains a list of those strokes that may overlap with the texel. Strokes are subject to scaling, up to a factor of two, as we interpolate between levels, so they must be rendered at their maximum size. The simplest way to gather seed IDs to texels is to render to a buffer of bit masks, where every bit indicates a seed, using *atomic bitwise or* operations. In a second pass, bit masks can be converted to ID lists, which require fewer bits for all but the smallest seed sets.

 ISBN 978-80-86943-70-1

DST	TAM	B16	C16	B64	C64
0.54	0.28	4.80	1.66	16.6	1.81

Table 1: Rendering time for a frame at 1920×1200 in ms.

The texture only needs to be updated if the artistic parameters are changed. The shading algorithm can be simply upgraded to query the seed coverage texture at $\mathbf{x}_s$, and only loop over the relevant seeds.

7 RESULTS

7.1 Performance

We have tested the algorithm on an NVIDIA GeForce GTX 780, with 1920×1200 full-screen resolution, and measured the time of rendering a single frame. We compared our solution against dynamic solid textures (DST) [BBT09] and Tonal Art Maps (TAM) [PHWF01], using code published by their authors. For RPTAM, we used 4 layers with 16 seeds each (B16 and C16 for brute force and coverage texture implementations, respectively, Figure 11), and with 64 seeds each (B64, C64, Figure 12). We used grayscale textures in all methods. Results in Table 7.1 show that even the brute force implementation delivers real-time performance, and all other methods have practically negligible rendering cost, with texture access bandwidth dominating. While the brute force method scales linearly with the number of seeds, the coverage texture acceleration eliminates this problem. While RPTAM is still slower than TAM, the difference has no significance on current hardware.

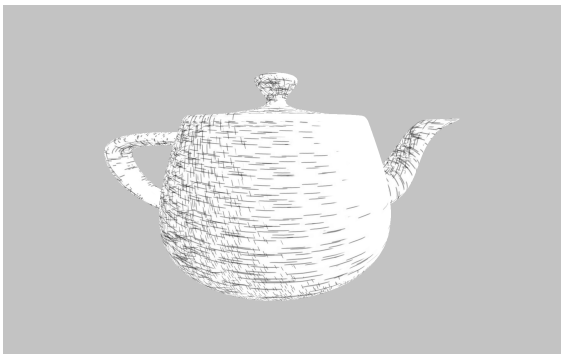

Figure 11: Teapot rendered with 4 tone layers, 16 seeds per layer, at 600 FPS, 1920×1200.

7.2 Quality

Our method, like dynamic solid textures and Tonal Art Maps, provides impeccable temporal coherence. Dynamic solid textures can produce *binary style* rendering to approximate hatching, but our method can work with stylized hatching strokes. Tonal Art Maps are equivalent to our approach in quality, but they do not offer infinite zooming. TAMs can be edited manually, or generated automatically by randomly inserting

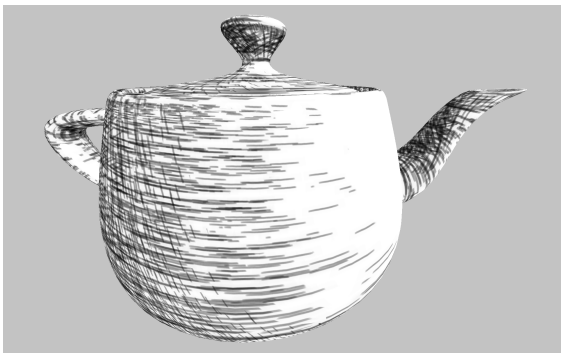

Figure 12: Teapot rendered with 4 tone layers, 64 seeds per layer, at 500 FPS, 1920×1200.

new strokes, and rejecting or clipping those colliding with existing ones. Both the TAM generation method and our seed set generation are lengthy trial-and-error searches, aiming at the same quality criteria of uniformly distributed strokes. Our method guarantees uniformity in a geometric sense. More importantly, as opposed to TAMs, seed sets do not need to be re-generated if artistic parameters like stroke length or stroke texture change, even allowing these to be animated. We can fade strokes simultaneously, producing rendering similar to TAMs (Figure 13), but also individually (Figure 14), which is a unique feature among texture-based hatching methods.

Figure 13: Teapot rendered with simultaneous stroke fading.

Figure 14: Teapot rendered with individual stroke fading.

8 FUTURE WORK

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 a slight periodic modulation in style is observable for simultaneous modulation. Therefore, we would like to expand the concept of nesting to weighted seed sets, with interpolated set featuring the same distribution of weights, making integer level stroke patterns 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.

9 ACKNOWLEDGEMENTS

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

10 REFERENCES

[AWI*09] AlMeraj Z., Wyvill B., Isenberg T., Gooch A. A., Guy R.: Automatically mimicking unique hand-drawn pencil lines. *Computers & Graphics 33*, 4 (2009), 496–508.

[BBT09] Bénard P., Bousseau A., Thollot J.: Dynamic solid textures for real-time coherent stylization. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), ACM, pp. 121–127.

[BD85] Barnsley M. F., Demko S.: Iterated function systems and the global construction of fractals. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences 399*, 1817 (1985), 243–275.

[BV11] Barnsley M. F., Vince A.: The chaos game on a general iterated function system. *Ergodic Theory and Dynamical Systems 31*, 04 (2011), 1073–1079.

[CRL01] Cornish D., Rowan A., Luebke D.: View-dependent particles for interactive non-photorealistic rendering. In *Graphics interface* (2001), vol. 1, pp. 151–158.

[Hae90] Haeberli P.: Paint by numbers: Abstract image representations. In *ACM SIGGRAPH Computer Graphics* (1990), vol. 24, ACM, pp. 207–214.

[Hal64] Halton J. H.: Algorithm 247: Radical-inverse quasi-random point sequence. *Communications of the ACM 7*, 12 (1964), 701–702.

[HZ00] Hertzmann A., Zorin D.: Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 517–526.

[JEGPO02] Jodoin P.-M., Epstein E., Granger-Piché M., Ostromoukhov V.: Hatching by example: a statistical approach. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering* (2002), ACM, pp. 29–36.

[LKL06] Lee H., Kwon S., Lee S.: Real-time pencil rendering. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (2006), ACM, pp. 37–45.

[Mei96] Meier B. J.: Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), ACM, pp. 477–484.

[PBPS09] Paiva A., Brazil E. V., Petronetto F., Sousa M. C.: Fluid-based hatching for tone mapping in line illustrations. *The Visual Computer 25*, 5-7 (2009), 519–527.

[PHWF01] Praun E., Hoppe H., Webb M., Finkelstein A.: Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, p. 581.

[Spi09] Spitzner D.: *Searchable Randomness*. Tech. Rep. 09-01, Department of Statistics, University of Virginia, January 2009.

[SS02] Strothotte T., Schlechtweg S.: *Non-photorealistic computer graphics: modeling, rendering, and animation*. Elsevier, 2002.

[USSK11] Umenhoffer T., Szécsi L., Szirmay-Kalos L.: Hatching for motion picture production. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 533–542.

[WH94] Witkin A. P., Heckbert P. S.: Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 269–277.

[Wil83] Williams L.: Pyramidal parametrics. In *ACM Siggraph Computer Graphics* (1983), vol. 17, ACM, pp. 1–11.

[WS94] Winkenbach G., Salesin D. H.: Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), ACM, pp. 91–100.

[ZISS04] Zander J., Isenberg T., Schlechtweg S., Strothotte T.: High quality hatching. In *Computer Graphics Forum* (2004), vol. 23, Wiley Online Library, pp. 421–430.