DOI: 10.1111/cgf.12055
EUROGRAPHICS 2013 / I. Navazo, P. Poulin
(Guest Editors)

Volume 32 (2013), Number 2

By–example synthesis of curvilinear structured patterns

Shizhe Zhou[1,2] Anass Lasram[1] Sylvain Lefebvre[1]

[1]INRIA, France [2]University of Science and Technology of China

Figure 1: *Our synthesizer generates a consistent, aperiodic pattern along a ribbon, from an example (red). It does not produce distortions in high curvature regions. The heart shape as well as the two right most results are closed curves.*

Abstract

Many algorithms in Computer Graphics require to synthesize a pattern along a curve. This is for instance the case with line stylization, to decorate objects with elaborate patterns (chains, laces, scratches), or to synthesize curvilinear features such as mountain ridges, rivers or roads.
We describe a simple yet effective method for this problem. Our method addresses the main challenge of maintaining the continuity of the pattern while following the curve. It allows some freedom to the synthesized pattern: It may locally diverge from the curve so as to allow for a more natural global result. This also lets the pattern escape areas of overlaps or fold-overs. This makes our method particularly well suited to structured, detailed patterns following complex curves.
Our synthesizer copies tilted pieces of the exemplar along the curve, following its orientation. The result is optimized through a shortest path search, with dynamic programming. We speed up the process by an efficient parallel implementation. Finally, since discontinuities may always remain we propose an optional post-processing step optimally deforming neighboring pieces to smooth the transitions.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

Many Computer Graphics approaches require to synthesize textured patterns along curves. This is for instance a key ingredient of line stylization [BCGF10, LYFD12] and structure propagation for image completion [SYJS05]. This is also useful to synthesize curvilinear features such as mountain ridges [ZSTR07], roads [GPGB11], rivers [SPK10], or to decorate meshes with details (bracelets, necklaces, illustrations, hatches) [KMM*02].

Usually, curves are texture mapped by exploiting their parametric domain. A ribbon is created by offsetting the curve. A texture is repeated inside while following curvature and orientation. The texture coordinates are the arc length parameterization and the distance to the curve. This however results in two artifacts: First, distortions introduced by curvature and second, pattern repetitions without any variation. While strict repetition is sometimes desirable, often the user

wishes a more natural, less regular aspect. Discontinuities may also appear if the pattern is not perfectly cyclic.

Our approach solves these issues by *synthesizing* a texture specifically for the curve. Distortions are avoided by working in the canvas space rather than in the parametric domain, and the synthesizer introduces variations in the result.

Several methods are related to this problem. [SYJS05] fills a missing part of an image by first synthesizing content along curves. Square overlapping patches are copied along a trajectory at regular intervals. The synthesized content does not follow the orientation of the curve. Other works synthesize content while following orientation [ZSTR07, SPK10, BCGF10]. However, they rely on stochastic synthesizers [WLKT09] which perform well on brush–like patterns, but fail to preserve the geometric continuity of structured patterns.

Several approaches synthesize curves from examples [HOCS02, MM10]. This is however different from our goal since we seek to synthesize a textured pattern rather than a single curved line.

Painterly rendering relies on brushes applied along curves [Her98, XLSN10], often following the silhouettes of a 3D object on screen [NM00, GVH07]. In this context, methods have also been proposed to distribute discrete elements in space or along curves, following an example [BBT*06, IMIM08]. *Helping hand* [LYFD12] stylizes a stroke by synthesizing along its trajectory a sequence of hand pose information – velocity, pressure and orientation. The example data is taken from a database of strokes. The synthesized information drives a virtual brush, producing a texture along the curve. Our problem is different: We focus on the synthesis of a structured pattern which is likely not the result of applying a brush along a path.

From a technical point of view our method is closer to the works of Sun et al. [SYJS05] and Lefebvre et al. [LHL10]. Both achieve high quality synthesis of structured textures by formulating a uni–directional problem which can be globally and exactly optimized. This is better suited to structured patterns: Because they require precise alignments, correct configurations have a much lower probability than with stochastic patterns.

Contributions: We propose a novel formulation for synthesizing patterns along curves which operates directly in the canvas space. Our approach produces new patterns without distorting the example content. The output of the synthesizer is a set of texture coordinates, and therefore does not require to store the result as an image. We relax the synthesis process by allowing some freedom to the pattern around the curve. This affords for more flexibility and lets the synthesized pattern exit problematic areas such as fold-overs due to strong local curvature. We cast the optimization process as a global optimization: a shortest path search conveniently solved using dynamic programming, and further accelerated by the GPU. Finally, we propose an optional post–processing to reduce any remaining discontinuities along the result pattern.

Input exemplars, comparing pixels: The input to our synthesizer is a pattern meant to be synthesized along a curve. The input is an image with a transparent background, which outlines the *geometry* of the pattern. During synthesis, when pixels are compared a larger importance is given to the geometry of the pattern, since it is most desirable to preserve its continuity. Therefore, there is a high penalty when matching opaque with transparent pixels.

We compare two RGB pixels p, q with values in $[0,255]^3$ with the following matching cost function:

$$\xi(p,q) = \begin{cases} ||p-q|| & \text{if } \alpha(p)\cdot\alpha(q) = 1 \\ ||p-q|| & \text{if } (1-\alpha(p))\cdot(1-\alpha(q)) = 1 \\ \gamma & \text{otherwise} \end{cases}$$

where $\alpha(.)$ extracts the transparency information of a pixel (1 if opaque, 0 otherwise). γ penalizes matching opaque and transparent pixels. We use $\gamma = 1024$. This could be adapted with a continuous cost in case of semi-transparent pixels. A more elaborate matching could also be obtained by applying an appearance space transform to the exemplar prior to synthesis [LH06].

1. Synthesizing patterns within a ribbon

1.1. Notations

Le E be the exemplar of size $W \times H$ and $\mathcal{C}$ the curve along which to synthesize. We assume a curve parameterized in arc length and denote L its total arc length. We denote by $\mathcal{C}(x), x \in [0..L]$ the points along the curve. Finally, we denote by $\mathbf{n}(x)$ the normal to the curve in $\mathcal{C}(x)$. We consider a ribbon around the curve. The ribbon is obtained by defining a top and bottom offset curves respectively as $\mathcal{T}(x) = \mathcal{C}(x) + \frac{b}{2}\mathbf{n}(x)$ and $\mathcal{B}(x) = \mathcal{C}(x) - \frac{b}{2}\mathbf{n}(x)$ with b the ribbon width. Note that we restrict ourselves to the 2D case for clarity, but the flat ribbon could also be defined in 3D. Similarly, the width b could vary freely along the ribbon if desired.

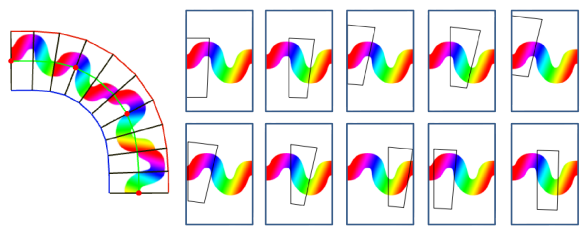


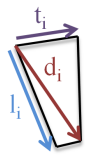
Figure 2: Left: *A synthesis result and the ribbon cage used to synthesize it. The ribbon is sliced into pieces along its medial axis.* Right: *Each piece of the ribbon is positioned into the exemplar so as to produce a visually continuous pattern. Since the left edge of each piece remains straight in the exemplar, no distortion appears along the curved ribbon.*

1.2. Overview

We initiate the synthesis process by slicing the ribbon into a number of pieces: Quadrilaterals living along the curve and connected through their left/right edges. Our synthesizer finds the best part of the exemplar to appear inside each piece – this is akin to finding texture coordinates for each piece in the exemplar. In this process, the left edge of a piece is always kept vertical in the exemplar, while we search for a position matching colors along the tilted right edge of the previous piece. The fact that we 'reset' the rotation at every piece allows the exemplar to smoothly follow the curvature. Since the pieces always exactly copy a part of the exemplar, there is no distortion. This process is illustrated in Figure 2.

1.3. Formulation

We generate pieces by slicing along the curve, with a sampling distance δ fixed by the user. Each piece is created as a quadrilateral with points $(\mathcal{T}(x_i), \mathcal{B}(x_i), \mathcal{B}(x_{i+1}), \mathcal{T}(x_{i+1}))$ for all $x_i = i\delta$ with $i \in [0..N]$, with $N = \frac{L}{\delta}$. For simplicity we assume that N is an integer – if not, a final smaller piece can be added to account for the fractional part. We rewrite the coordinates as $(\mathcal{T}(x_i), \mathcal{T}(x_i)+\mathbf{l}_i, \mathcal{T}(x_i)+\mathbf{d}_i, \mathcal{T}(x_i)+\mathbf{t}_i)$ where $\mathbf{l}_i$, $\mathbf{d}_i$, $\mathbf{t}_i$ are vectors computed between the top left corner and the other corners, as illustrated in the inset. In areas of high-curvature pieces may fold over themselves, the right edge crossing the left edge. This can be observed in Figure 3. We treat these cases by forbidding the pattern to appear in the folded area, using the method described in Section 3.1. The work of Asente [Ase10] could also be used to slice the curve without folding.



Each piece gets coordinates in the exemplar. Because we do not stretch the pieces, only the position of the top left corner needs to be fixed. In the exemplar each piece is oriented so that its left edge is vertical, while the right edge may be tilted due to the curvature (see Figure 2). We note u_i the coordinates of the top left corner in E. We note $\mathbf{y}$ the vertical direction $(0,-1)$ and M the matrix transforming $\mathbf{l}_i$ into $\mathbf{y}$. A scaling factor s controls the relative size of the synthesized pattern with respect to the curve.

Following these notations, we write the piece polygon in E as $(u_i, u_i + s\cdot\mathbf{y}, u_i + s\cdot M\mathbf{d}_i, u_i + s\cdot M\mathbf{t}_i)$. We note the left edge as $L_i = (0, s\cdot\mathbf{y})$ and the right edge as $R_i = (s\cdot M\mathbf{t}_i, s\cdot M\mathbf{d}_i)$. In the following, we note a translation of an edge L by a vector u as $u+L$.

The set of coordinates u_i are the variables of our problem. A good selection of coordinates across all pieces will produce a continuous pattern within the ribbon. We define the cost of a given choice of u_i for all pieces as:

$$\Omega(u_0,\ldots,u_N) = \sum_{i=0}^{N-1} \mathcal{D}(u_i+R_i, u_{i+1}+L_{i+1})$$

where $\mathcal{D}$ measures the distance of the colors along the edges between pieces. $u_i + R_i$ is the right edge of piece i positioned at u_i. Similarly, L_{i+1} is the left edge of the following piece positioned at u_{i+1} in the exemplar. Ω is thus the cost of matches for all right/left edge pairs along the curve. Closed curves require a special treatment described in Section 1.5.

$\mathcal{D}$ is computed by extracting part of a column in E for the left edge L_{i+1} and by sampling along the tilted right edge R_i. We note $C = \lceil ||L_i|| \rceil$ the number of samples taken along the edges. Two color vectors of size C are obtained. We note these vectors $\bar{R}_i$ and $\bar{L}_{i+1}$, and index them with the usual array notation. The two vectors are then compared with a sum of per-pixel matching costs:

$$\mathcal{D}(u_i+R_i, u_{i+1}+L_{i+1}) = \sum_{k=0}^{C-1} \xi(\bar{R}_i[k], \bar{L}_{i+1}[k])$$

An important observation which makes this optimization much more efficient is that not all u_i are valid for a piece: In many positions the piece either is empty, or does not slice the pattern completely: The top or bottom edge intersects the pattern. This is illustrated in the inset, where only the green position is valid. By selecting only valid positions during the optimization we strongly reduce the search space (typically 5% to 35% of the positions are valid).

1.4. Optimization

We optimize for the best u_i by dynamic programming. We compute a table P in which an entry $P[i,u]$ is the best possible cost of using coordinates u for piece i. The recursive relationship is given as:

$$P[i,u] = \min_v (P[i-1,v] + \mathcal{D}(v+R_{i-1}, u+L_i)) \qquad (1)$$

In this process we only consider valid positions u for piece i and valid positions v for piece $i-1$. Table P is computed by dynamic programming (DP).

Complexity The DP table is sequentially updated, row by row, from top to bottom. The number of rows equals the number of pieces N. At each position along each row we search through all the positions of the prior piece (see equation 1) and compare two columns of size C. The number of positions that a piece can take depends the exemplar size and the column size C: In order to consider all possibilities the top left corner of a piece may be located up to C pixels above the exemplar. We denote the number of positions by $U = W \times (H+C)$. The asymptotic complexity of a naive algorithm is $O(NU^2C)$. This grows linearly with the number of pieces, but grows quadratically in the number of exemplar pixels. This large complexity is made practical thanks to several factors: Many computations can be parallelized, the number of *valid* positions is in practice much smaller than U, and the comparison of columns can be factored to reduce the complexity to $O(NU^2)$. Section 2 gives further details on how to implement the approach efficiently.

1.5. Closed curves

Closed curves are cut at an arbitrary location and treated as non-closed. A *terminal* last piece is added: It represents the connection back to the first piece. Only solutions where the position for the terminal piece equals the position for the first piece are valid (cycle).

This problem is akin to using DP to find a best cut around a patch, searching for a cyclic path in a ring [JSTS06]. Finding the optimal cycle requires running several DPs, one per choice for the start location. However, it can be well approximated with a single unconstrained DP [LL12]. We follow a similar approach and search for a cycle by backtracking from all terminal positions after running the DP optimization once. Figure 1 (right) illustrates synthesis along a closed curve. Alternatively, results where the ending position is vertically within a threshold of the starting position may be used with the stitching described in Section 3.2. We did not use this approach in our results.

2. Implementation

Our implementation is based on OpenCL and designed to run on the GPU. The DP algorithm is as follows:

```
findValidPositions( 0 )
for (int i = 1 ; i < N ; i ++ ) {
  extractColumns( i - 1 )
  findValidPositions( i )
  compactValidPositions( i )
  computeCostTable( i )
  updateTableRow( i )
}
```

Each iteration of the for loop fills a row of the DP table. Each of the function corresponds to a kernel call. The loop starts at $i = 1$ since the first piece is unconstrained – unless otherwise specified by the user. After iterating over all pieces we backtrack from the bottom of the table to the top, effectively finding the optimal sequence of positions of the pieces.

`extractColumns` samples colors along the right–edges of piece $i-1$, gathering $\bar{R}_{i-1}$ for all *valid* positions of piece $i-1$. The kernel is launched with one thread per column.

`findValidPositions` computes all valid positions for the i-th piece. This involves, for each of the U possible positions in the exemplar, testing whether the piece would be in a valid configuration. The kernel is launched with one thread per position. The output is an array of booleans flagging which positions are valid. Validity is tested by sampling along the edges and the inside of the piece, checking that 1) the piece correctly slices the exemplar pattern (see Section 1.3) and 2) the piece does not overlap a forbidden region of the canvas (see Section 3.1).

`compactValidPositions` gathers only the valid positions for the piece with a parallel scan operation. An indexing array is created to remap the valid indexes in a continuous, smaller table. Typically only between 5% and 35% of the positions are valid. This offers a significant reduction in memory and computational requirements.

`computeCostTable` compares all left edges of piece i at valid positions, to the right edges of piece $i-1$ gathered earlier. The result is a cost table containing the value of $\mathcal{D}$ for all right/left edge pairs. The kernel is launched with one thread per column-pair, each comparing C pixels.

`updateTableRow` updates the DP table following Equation 1. The kernel is launched with one thread per row entry.

Discussion The most expensive step is `computeCostTable`, with a complexity of $O(U^2C)$. This can be reduced to $O(U^2)$ by relying on a sliding accumulation window. The error for a first pair of columns is computed in an accumulation window of C pixels. The window is then moved down by one pixel, computing the error for the neighboring pair. This only requires removing/adding the error of the two pixels exiting/entering the window, canceling the C factor in the complexity. This process however assumes a continuous image when in fact we consider only *valid* positions, a sparse subset of U positions. We nevertheless experimented with this approach, comparing the exemplar and a transformed version of it accounting for the tilted angle of the previous piece edge. We only store the result for pairs of columns which are on valid positions. In practice this results in a speed-up when C becomes larger than H (a ribbon wider than the pattern). However, in the most typical case where C and H have similar values, the overhead is significant and the naive comparison of columns remains more efficient. This is due to the sparsity of the valid positions which compensates for the overhead of the C comparisons.

3. Control

3.1. Region avoidance

Our optimizer supports region avoidance: We invalidate coordinates that would make the pattern collide with forbidden areas in the canvas. This is done by testing, for every piece position in the exemplar, whether it makes an opaque pixel fall within the forbidden regions. If it is the case, the position is flagged as invalid (see Section 1.3). The forbidden regions are described in an image which covers the entire canvas. We call this image the *avoidance map*.

A specific case of region avoidance is foldovers of the ribbon, due to high curvature or close proximity. For these cases the avoidance map is obtained by drawing in an off-screen buffer, flagging regions of overdraw where the curve self-overlaps. Problematic areas are shown in red Figure 3.

One limitation of this approach is that the foldover area is strictly excluded, when in fact it could be used once by the pattern. This could be partially addressed by checking at every DP step whether the current sub–path already uses the foldover area. However, this would be expensive to compute.

3.2. Stitching

Even though our results are globally optimal, visible seams may remain: It is rare that the exemplar offers degrees of

freedom allowing for perfect matches between pieces. In addition, a straight exemplar necessarily produces discontinuities when synthesized along a curve; the tilted edges have a different length than the vertical edges.

We further improve the results by deforming the pieces. We compute an optimal alignment of the right edge of piece i with the left edge of piece $i+1$. The deformation is then propagated to the piece interior. Figure 4 compares a result without and with stitching.

Alignment: We compute a remapping of the indexes of the pixels along the right edge so that they better match those along the left edge. This is done by computing a table M so that $M[s,t]$ contains the best possible cost resulting of having pixel t of the right edge in front of pixel s of the left edge. The table is computed by dynamic programming with the following recursive relationship:

$$M[s,t] = \min \begin{cases} M[s-1,t-1] \\ M[s-1,t] + \beta(\bar{L}_{i+1}[s], \bar{R}_i[t]) \\ M[s,t-1] + \beta(\bar{L}_{i+1}[s], \bar{R}_i[t]) \end{cases} + ||\bar{L}_{i+1}[s] - \bar{R}_i[t]|| \quad (2)$$

where $\beta(.,.)$ penalizes the distortion of opaque segments:

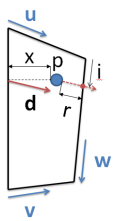
$$\beta(L,R) = \begin{cases} P & \text{if } \alpha(\bar{L}_{i+1}[s]) > 0 \\ P & \text{if } \alpha(\bar{R}_{i+1}[s]) > 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

In practice we use $P = 32$ (RGB colors are within $[0,255]^3$).

By backtracking in M we obtain an array remapping the pixel indexes along the right edge. We note this array T. In absence of deformation, $T[i] = i$. We compute all tables for all pieces in parallel, each GPU thread handling one piece.

Propagation: To obtain a continuous deformation we interpolate from the edge to the inside of the piece. This is done during rendering by the pixel shader and requires no other storage than the arrays T of the pieces.

The principle for propagating the deformation is illustrated in the inset. $\mathbf{u}, \mathbf{v}$ and $\mathbf{w}$ are the direction vectors of, respectively, the top, bottom and right edges. Since we are working in the exemplar space, the left edge is vertical (see Section 1.2). Our goal is to compute a deformation of the texture coordinates at the sampling point p. We first interpolate the direction $\mathbf{d}$ from $\mathbf{u}$ and $\mathbf{v}$, using the vertical position of p. We then project p along $\mathbf{d}$ onto the right edge. This computes the distance i along the edge. We use this distance to obtain a deformation as $\Delta = T[i] - i$. The final deformation to the texture coordinates is $\frac{x}{x+r} \cdot \Delta \cdot \mathbf{w}$. The term $\frac{x}{x+r}$ controls the strength of the deformation between the left/right edges: The deformation is largest along the right edge, and has no influence along the left edge.



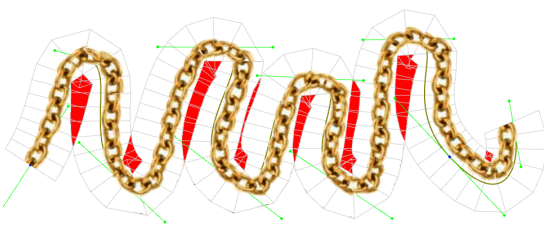


Figure 3: *A chain is synthesized while avoiding foldovers and overlaps (in red).*

Figure 4: Left: *Straight patterns produce discontinuities in curved regions.* Right: *Our stitching step removes most of the discontinuities at little memory overhead during display.*

Figure 5: Top left: *Structured exemplar.* Bottom left: *Result of Bénard et al. [BCGF10]. Note the loss of continuity in the pattern.* Right: *Part of our result along a curve.*

4. Results

Figures 1 and 7 show a variety of results for different patterns. Note how the pattern continuity is preserved even in high-curvature areas. The synthesized patterns also exhibit variety along the curve, producing a more natural result than a strict repetition. Figure 5 shows a comparison with the scheme of Bénard et al., for the case of a structured pattern.

All results are obtained on a NVidia GeForce GTX 580. Adding a piece to the curve – computing one row of the DP table – takes from 30 ms to 650 ms for exemplar sizes of respectively 199×82 and 325×64. Overall performance for a curve depends on the number of pieces. Our results took between a few seconds and 30 seconds. Figure 4 illustrates the performance behaviour when increasing the number of pieces or the exemplar width. This is consistent with the asymptotic complexity (Section 1.4).

Limitations: A difficulty inherent to our formulation is the dependency of the result on the length δ and width b of the pieces. In particular, short pieces can quickly lead to undesirable repetitions. This could be avoided through histogram constraints [LHL10]. We do not strictly enforce continuity: The optimizer may choose to break the pattern in difficult situations. Strictly enforcing continuity however can quickly exhaust all possible solutions. This is an area for further study. The computational cost currently limits the maximum resolution of the exemplar. However, it is possible to optimize at a low resolution and reuse the results on higher res-

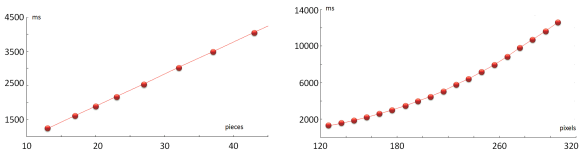


Figure 6: Left: *Performance (ms) versus number of pieces (N), with* $W \times H = 156 \times 64$ *and* $b = 90$. *Note the linear behavior.* Right: *Performance (ms) versus exemplar width (W).* $H = 64, N = 23, b = 90$. *Note the quadratic increase.*

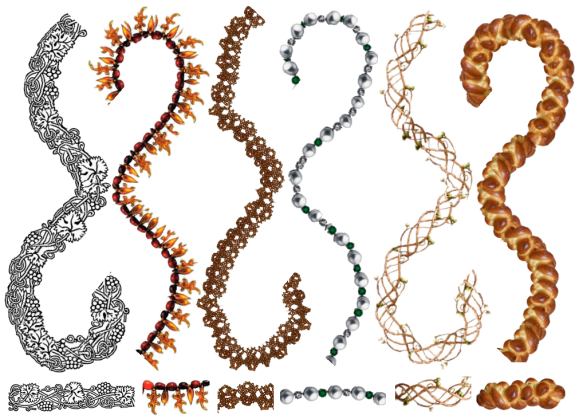


Figure 7: *Various synthesized patterns. Exemplars are shown at the bottom. None of them are cyclic.*

olution images. We did not used this approach on our results. Another improvement is to consider incremental updates to the curve during user manipulation. Many computations could be reused in this situation.

5. Conclusion

We synthesize patterns along curves by copying undistorted pieces from an exemplar. Our synthesizer freely grows the pattern in the vicinity of the curve, letting the pattern exit problematic areas. Thanks to a parallel implementation on the GPU, results are produced in seconds. Our optional stitching step aligns edges and propagates deformations at render time: Only the exemplar, the piece coordinates and the edge deformations have to be stored for rendering.

There are many ways to adapt this work to other settings, by augmenting the objective cost with additional terms. For instance, one may wish to take into account a stochastic background texture instead of a transparent background. Another possibility is to mix different input patterns during synthesis, allowing matches across exemplars.

6. Acknowledgments

This work was funded by the Agence Nationale de la Recherche 2008-COORD-021-01. Many images are from Flickr users: (Fig.1) *graceExtremiss, takanaImg416, michael pollak, GrandmaMarilyns, Social Butterfly Jewellery, james_gordon_losangeles.* (Fig.7) *graceExtremiss, Bellafaye, CircaSassy, OhDarkDevil.* (Fig.4) *Crossett Library Bennington College.*

References

[Ase10] ASENTE P.: Folding avoidance in skeletal strokes. In *Proceedings of the Seventh Sketch-Based Interfaces and Modeling Symposium* (2010). 3

[BBT*06] BARLA P., BRESLAV S., THOLLOT J., SILLION F., MARKOSIAN L.: Stroke pattern analysis and synthesis. In *Computer Graphics Forum* (2006), vol. 25. 2

[BCGF10] BÉNARD P., COLE F., GOLOVINSKIY A., FINKELSTEIN A.: Self-Similar Texture for Coherent Line Stylization. In *NPAR* (2010). 1, 2, 5

[GPGB11] GALIN E., PEYTAVIE A., GUERIN E., BENES B.: Authoring hierarchical road networks. *Computer Graphics Forum 29* (2011). 1

[GVH07] GOODWIN T., VOLLICK I., HERTZMANN A.: Isophote distance: a shading approach to artistic stroke thickness. In *NPAR* (2007). 2

[Her98] HERTZMANN A.: Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH* (1998). 2

[HOCS02] HERTZMANN A., OLIVER N., CURLESS B., SEITZ S. M.: Curve analogies. In *the Eurographics Workshop on Rendering* (2002). 2

[IMIM08] IJIRI T., MECH R., IGARASHI T., MILLER G.: An example-based procedural system for element arrangement. In *Computer Graphics Forum* (2008), vol. 27. 2

[JSTS06] JIA J., SUN J., TANG C.-K., SHUM H.-Y.: Drag-and-drop pasting. *ACM Transactions on Graphics 25*, 3 (2006). 4

[KMM*02] KALNINS R., MARKOSIAN L., MEIER B., KOWALSKI M., LEE J., DAVIDSON P., WEBB M., HUGHES J., FINKELSTEIN A.: Wysiwyg npr: Drawing strokes directly on 3d models. In *ACM Transactions on Graphics* (2002), vol. 21. 1

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. *ACM Transactions on Graphics 25*, 3 (2006). 2

[LHL10] LEFEBVRE S., HORNUS S., LASRAM A.: By-example synthesis of architectural textures. *ACM Transactions on Graphics 29*, 4 (2010). 2, 5

[LL12] LASRAM A., LEFEBVRE S.: Parallel patch based texture synthesis. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics* (2012). 4

[LYFD12] LU J., YU F., FINKELSTEIN A., DIVERDI S.: HelpingHand: Example-based stroke stylization. In *ACM Transactions on Graphics* (2012). 1, 2

[MM10] MERRELL P., MANOCHA D.: Example-based curve generation. *Computers & Graphics 34* (2010). 2

[NM00] NORTHRUP J., MARKOSIAN L.: Artistic silhouettes: A hybrid approach. In *NPAR* (2000). 2

[SPK10] SIBBING D., PAVIC D., KOBBELT L.: Image synthesis for branching structures. *Computer Graphics Forum 29*, 7 (2010). 1, 2

[SYJS05] SUN J., YUAN L., JIA J., SHUM H.-Y.: Image completion with structure propagation. In *SIGGRAPH* (2005). 1, 2

[WLKT09] WEI L.-Y., LEFEBVRE S., KWATRA V., TURK G.: State of the art in example-based texture synthesis. In *Eurographics STAR* (2009). 2

[XLSN10] XIE N., LAGA H., SAITO S., NAKAJIMA M.: Ir2s: interactive real photo to sumi-e. In *NPAR* (2010). 2

[ZSTR07] ZHOU H., SUN J., TURK G., REHG J. M.: Terrain synthesis from digital elevation models. *Transactions on Visualization and Computer Graphics 13*, 4 (2007). 1, 2