Pacific Graphics 2012
C. Bregler, P. Sander, and M. Wimmer
(Guest Editors)

Volume 31 (2012), Number 7

Multi-scale Assemblage for Procedural Texturing

G. Gilet[1], J-M. Dischler[2] and D. Ghazanfarpour[1]

[1]XLIM - UMR CNRS 7252, University of Limoges, France
[2]LSIIT - UMR CNRS 7005, University of Strasbourg, France

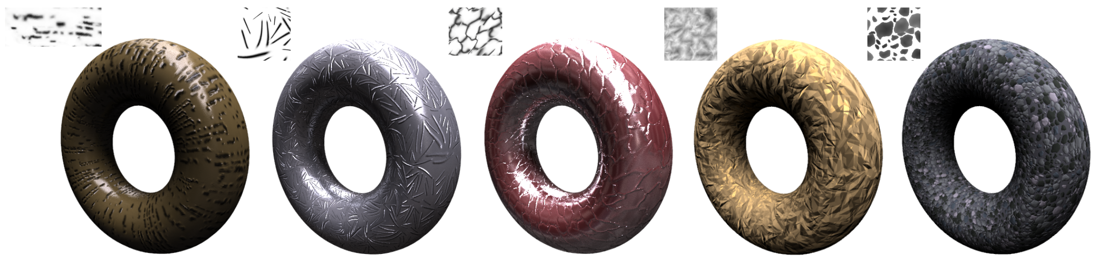


Figure 1: *Multi-scale assemblage is a random pattern generation process generalizing sparse convolution. It allows users to design interactively new types of texture basis functions (noise-like functions) and / or structured patterns by preserving all advantages of procedural definitions, namely infinity without repetition, definition independency and extreme compactness. These textures require no texture memory and fit entirely into the shader program.*

Abstract

A procedural pattern generation process, called multi-scale "assemblage" is introduced. An assemblage is defined as a multi-scale composition of "multi-variate" statistical figures, that can be kernel functions for defining noise-like texture basis functions, or that can be patterns for defining structured procedural textures. This paper presents two main contributions: 1) a new procedural random point distribution function, that, unlike point jittering, allow us to take into account some spatial dependencies among figures and 2) a "multi-variate" approach that, instead of defining finite sets of constant figures, allows us to generate nearly infinite variations of figures on-the-fly. For both, we use a "statistical shape model", which is a representation of shape variations. Thanks to a direct GPU implementation, assemblage textures can be used to generate new classes of procedural textures for real-time rendering by preserving all characteristics of usual procedural textures*, namely: infinity, definition independency (provided the figures are also definition independent) and extreme compactness.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: procedural texturing, noise, procedural object distribution function, GPU shader

1. Introduction

Content creators tend to design larger and larger virtual worlds characterized by a huge amount of visual details, which are commonly modeled using textures. However, while increasing texture complexity improves visual quality, it also raises more and more problems concerning: 1) excessive storage requirements and 2) prohibitive synthesis timings. Many popular texture synthesis techniques, like synthesis "by example" [WLKT09] or "physical simulation" [DRS08] do not scale well, since the memory consumption as well as the computational complexity grow proportionally w.r.t the surface size and the texture definition. *Procedural textures* [EMP*98] intrinsically avoid the two previous problems. Instead of fetching texture values from massive pre-computed data arrays, a little program directly computes the values independently for all coordinates of parameterization space at constant time complexity. Since there is no need to reference previous calculations, procedural textures are not depending on the surface size. Their most interesting advantage is the ability to generate visual complexity with quasi-infinite variation at arbitrary definition while requiring only a marginal memory cost. Impressive GPU improvements could furthermore explain the recent renewed interest for these kinds of textures. In particular, the study of [OHL*08] outlines that, during the past decade, GPU computational power grew very fast, whereas GPU memory bandwidth did not grow at the same rate. Thus, putting efforts on "on-the-fly" computations instead of "out-of-core" memory management seems promising for managing virtual scenes with ever-increasing visual complexity.

Unfortunately, the creation of procedural textures is not a simple task. The classes of patterns that can be represented is limited by the narrow range of available *procedural basis functions*, the most common ones being noise and turbulence (a sum of noises) [Per85]. As only low-order statistics (related to the spectral energy) are considered, many kind of structured stochastic patterns cannot be *directly* modeled using these functions. Procedural basis functions must often be combined with other mathematical functions to produce structured patterns. For example, [Per85] combines turbulence with a sine wave (resp. with splines) to create marble veins (resp. cloudy sky). Choosing the right combination of functions to produce a given visual result raises many difficulties for most users, as good mathematical and (often) programming knowledge becomes almost mandatory.

In that respect, our motivation is to propose a novel stochastic procedural pattern generation process (similarly to [RB85]) called multi-scale *assemblage*. The objective is to facilitate the creation of structured stochastic procedural patterns (noise-like basis functions and textures) by extending the simpler principle of *sparse convolution*. The terminology of assemblage has been borrowed from an artistic process, which consists in creating art compositions by putting together diverse objects. Multi-scale assemblage is based on two contributions: a new procedural point distribution function and a dynamic stochastic figure generation process, a figure being a function (or an image) to which we apply the convolution operation. We propose to define our random point distribution function by using higher dimensioned topological primitives, like polygonal cells (2-D), instead of directly "jittering" points (0-D). The centers of the cells are used to define positions, while cell edges are used to set up dependencies among figures. Other spatial relations, such as alignments, are also addressed by using other primitives such as curves (1-D). The core issue is to be able to create infinite random variations of primitives and figures, without using repetition. To do so, we define a hierarchical *statistical shape model*, which allows us to represent shape variations, using statistical modes.

Our hierarchical extension allows us to compose complex figures on the basis of simpler randomly generated subfigures. Based on this model, an assemblage consists of hierarchical compositions of "multi variate" figures, positions and figures being computed from a linear combination of randomly weighted principal modes (multi variate means in our case that we use multiple independent random variables). The principal modes are "statistically learned" from user-edited sets of basis figures and primitives. All manipulations thus remain purely interactive and no mathematical / programming knowledge is required for users. Figure 1 shows examples of procedural patterns generated using our assemblage technique.

The rest of the paper is organized as follows. Section 2 gives a short overview of works related to procedural texturing. Section 3 introduces the hierarchical statistical shape model we use. Section 4 presents our point distribution functions. Section 5 describes our procedural multi-scale assemblage technique. In particular, we show how textures can be efficiently rendered at real-time rates using a GPU implementation. Before concluding, section 6 presents results and discusses limitations.

2. Related Works

Procedural textures, as defined in [EMP*98], should not be confused with the more general concept of "iterative procedures" (algorithms) for texture synthesis. In this specific context, *procedural* means that texture values are explicitly computed by functions at *run-time* instead of fetching them from data arrays (texture maps). An important property is: the functions are accessed at constant time and memory complexity, independently of previous calculations, for any random location of the *entire continuous and infinite space* without using periodicity. As corollary, memory consumption and synthesis timings become completely independent of surface size and texture definition. Generally, procedural textures use "quasi-infinite" stochastic *procedural basis functions*. These functions are not truly infinite since they are based on hash-coding. But because they have an extremely large period, they nevertheless can be considered as infinite for usual runs. The two main categories of basis functions are noise (see survey [LLC*10]) and random point distributions. Random point distributions are fundamental for procedural texturing, since they represent a support function for several other procedural basis functions: sparse convolution noises [Lew89, vW91, LLDD09], cellular noises computed using n-th closest distances [Wor96] and bombing patterns consisting of randomly "dropped" figures, that are for example textured quads in [Gla04]. Bombing has inspired a tremendous amount of texture synthesis techniques in computer graphics, "figures" being alternatively called bombs, sprites, textons, particles, etc. (see for instance [DMLG02, LHN05]). However, instead of using procedural *distribution functions*, most of these methods compute *ex-*

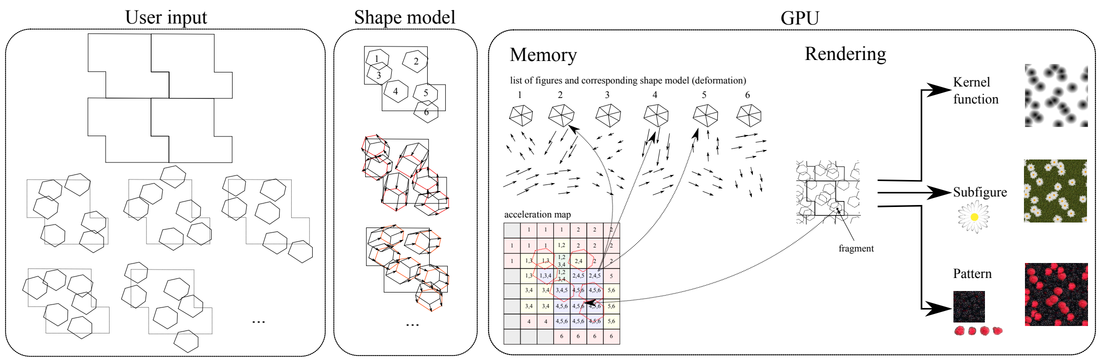


Figure 2: *Workflow of our method. Left: the user edits figures that are associated to cells (top) defining rectilinear tesselations of the plane. Middle: a statistical shape model is computed from the user input. Right: the resulting model is stored in GPU memory (geometric elements and sets of displacement vectors), along with an acceleration map to compute assemblage textures. The final rendering (right) uses either a distance function, sub-figures or patterns.*

plicit sets of figures and positions using more or less complex iterative (for instance relaxation-based [DZ06]) mechanisms.

Point distribution functions are *procedural* when they avoid the explicit storage of positions by generating infinite sets of points "on-the-fly". Jittering is the most common technique, as it is simple and fast (see GPU implementations of [Gla04, LLDD09]). However, it has the drawback of being unable to consider higher-order dependencies. [LD05] replaces jittering with a random process inspired by scanline stochastic tiling [CSHD03]. The process is called *direct stochastic tiling*. Some global spatial dependencies can be taken into account. In particular, [LD05] generates infinite and non-periodic point sets characterized by a Poisson-disk distribution, i.e. a distribution in which no two points are closer than a threshold distance. The drawback is that, unlike jittering, specific tiles must be pre-generated and stored. Because the pre-computed set of tiles is finite, some type of "repetition" necessarily occurs: the same tiles are re-used again and again at different locations. Our approach avoids repetition. As for jittering, we propose to generate random variations on-the-fly.

Manipulating stochastic basis functions and mathematical functions for modeling textures with desired visual characteristics is not straightforward in most cases. To make the modeling of procedural textures easier for users, mainly two categories of approaches have been proposed. The first one exploits the fact that large "texture shader" databases already exist [BD04, LLD12]. They mainly provide tools to improve shader parameter browsing. Such methods are however limited by the content of the available databases. The second category attempts to derive procedural textures by using example images [WLKT09, LVLD10, GDS10, GLLD12]. They consist in adjusting the parameters of underlying sums of noises by analyzing the power spectrum. Since noise functions, standing alone, do cover only a narrow range of low-order statistics, purely noise-based methods are limited to a narrow range of stochastic "micro-patterns". Our assemblage approach avoids this limitation by composing random figures / kernel functions in a "structured" way. The figures are generated hierarchically and dynamically while taking into account shape dependencies with neighbors. It can be considered as a kind of extension of *sparse convolution*, the latter being in our framework a particular case of single-scale assemblage with no dependencies among neighboring kernels.
The creation of visual variations of geometric objects is a vast and active research area, but most methods are specific to some types of parametric, for example grammar-based, objects like leaves [PTMG08] or buildings [MGHS11]. In our case, we use a *statistical shape model* combined with a simple random process. Random processes, especially Markov processes, are already widely used in computer graphics. See for example cell-structured / bombing models [SA79] for random pattern generation, stochastic subdivision for terrain synthesis [FFC82], random walks for texture synthesis "by example" [ZG02], etc. Conversely, the use of statistical shape models has not been yet applied, to the best of our knowledge, to procedural texture synthesis. In computer graphics, it has been used successfully to generate human body animations by "statistical learning" from 3D scans [ASK*05]. An overview of statistical shape models for vision and pattern recognition can be found in [DM98].

3. Hierarchical statistical shape models for dynamic figure synthesis

Figure 2 summarizes our approach for creating / editing procedural textures. In this section, we show how random variations of figures, that we will call *instances*, can be dynamically created (left and middle parts of figure 2). We first introduce statistical shape models for single-scale figures. Then, we extend our figure model to multi-scale definitions.

In a second step (next section), these figures are distributed more or less randomly over the infinite 2D plane (right part of figure 2) so as to create procedural textures by assemblage.

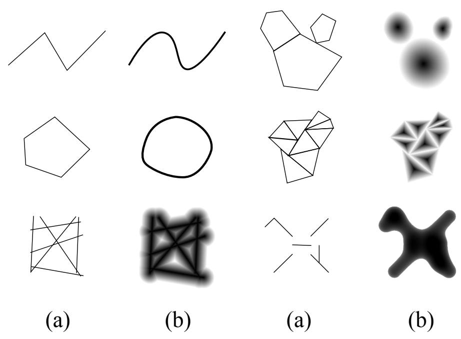


Figure 3: *Continuous figures (or kernel functions) are defined on the basis of sets of discrete geometric elements of 2D Euclidian space. (a) shows examples of elements: polygons and polylines. (b) examples of resulting continuous functions using distance computations and interpolations.*

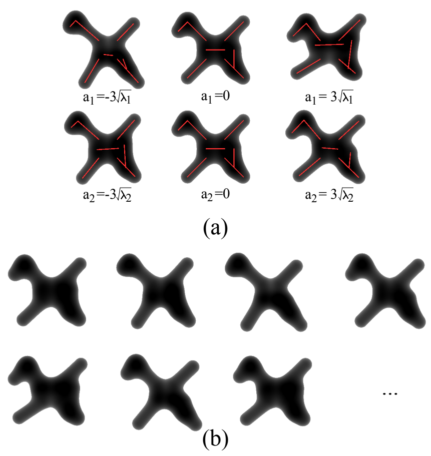


Figure 4: *Single-scale statistical shape model for creating random figure variations. (a) using the two first principal modes. (b) A few examples of corresponding random instances.*

3.1. Single-scale statistical figures

We consider that a figure Φ is defined on the basis of sets of discrete geometric elements of 2D Euclidian space: polygons and polylines that can be connected or isolated. A continuous figure or kernel function can be derived from these discrete elements using for instance splines, recursive subdivision schemes or potential (distance) functions for implicit modeling. We denote the resulting continuous figure $\mathcal{R}(\Phi)$. Some examples are shown in figure 3. For each set of elements Φ, we show a corresponding continuous function $\mathcal{R}(\Phi)$. We focus on statistical variations of elements, that will infer variations of $\mathcal{R}(\Phi)$, independently of the way these figures are made continuous w.r.t. their corresponding discrete elements. In the remaining parts of the paper, we use equally the term figure for the "set of discrete geometric elements" Φ as well as for the corresponding continuous representation $\mathcal{R}(\Phi)$.

Φ is defined by a set of vertices $\{v_i = (x_i, y_i)\}$, $i \in [1, n_V]$ and a set of edges, an edge e being a binary relation between vertices: $e = \{v_{\iota_1}, v_{\iota_2}\}$, $\iota_{1,2} \in [1, n_V]$ and $\iota_1 \neq \iota_2$.

To be able to compute any statistical shape model, we need multiple figure "samples" Φ_k, $k \in [1,N]$. These samples must have the same amount of edges and vertices so that they only differ by the vertices positions. In our case, all figures have been edited / painted by using an own interactive vector graphics drawing tool. Our tool allows one to create and modify polygons and polylines. The user first designs a basis figure and then creates the "samples" by interactively displacing the vertices. But a priori, any other commercial / free vector graphics tool could be used instead, provided the file format allows one to register correctly the figure samples that must effectively have the same amount of vertices. To be representative, the number of edited figures should not be too low. We used at least ten figure samples. A sample figure is thus defined by a vector of $2n_V$ elements $\Phi_k = [x_1^k, x_2^k, \cdots, x_{n_V}^k, y_1^k, y_2^k, \cdots, y_{n_V}^k]^T$. The corresponding shape space is then obtained by applying singular value decomposition (SVD). The goal is to express the input set of sample figures in a uncorrelated orthogonal frame defined by matrix $[\Psi_j]$, i.e. such that:

$$\Phi_k = \overline{\Phi} + \sum_{j=1}^{N} w_k [\Psi_j] \tag{1}$$

where

$$\overline{\Phi} = \frac{1}{N} \sum_{k=1}^{N} \Phi_k = [\overline{x}_1, \overline{x}_2, \cdots, \overline{x}_{n_V}, \overline{y}_1, \overline{y}_2, \cdots, \overline{y}_{n_V}]$$

and w_k is a vector of weights corresponding to sample figure Φ_k. The new frame is built from the covariance matrix:

$$\Sigma_\Phi = \frac{1}{N} \sum_{k=1}^{N} [\Phi_k - \overline{\Phi}][\Phi_k - \overline{\Phi}]^T$$

such that $\Sigma_\Phi = [\Psi_j] \Sigma_\Psi [\Psi_j]^T$, where Σ_Ψ represents the diagonal matrix of eigenvalues λ_j.

With this formulation, an eigenvector Ψ_j corresponds to a set of displacement vectors used to apply a deformation to the mean shape $\overline{\Phi}$. All original sample figures are expressed as weighted sums of displacement vectors applied to the mean figure, the displacements being ordered by decreasing deformation energy. The amplitude of deformation resulting from each Ψ_j can be evaluated by making the vector of weights vary at the corresponding eigenvalue λ_j, while setting all other weights to 0: $w = (0, \cdots, 0, a_j, 0, \cdots, 0)$.

For example, we can take $a_j \in \{-3\sqrt{\lambda_j}, +3\sqrt{\lambda_j}\}$. Modifying one component at a time defines the corresponding *mode*, the first mode representing the most dominant variations. We show examples of two principal modes in figure 4(a). The middle column of part (a) shows the mean figure, i.e. using $a_j = 0$. We exploit this statistical shape model to generate arbitrary random figures $\Phi_{(r)}$, that we call random *instances*, while preserving basic correlations w.r.t the figure's shape. We just have to choose random weights $w = (\xi_1, \cdots, \xi_M, 0, \cdots, 0)$ between some bounds, for example $\xi_j \in [-3\sqrt{\lambda_j}, +3\sqrt{\lambda_j}]$. Part (b) shows examples of random instances obtained using two principal modes only (the modes of part (a)). Since the eigenvalues decrease rapidly, a few principal modes $M << N$ generally turn out to be sufficient for defining variations. Of course the actual variety of random figures strongly depends on the variety of the user supplied samples. The more these samples are visually "different", the greater the variety.

3.2. Multi-scale figures

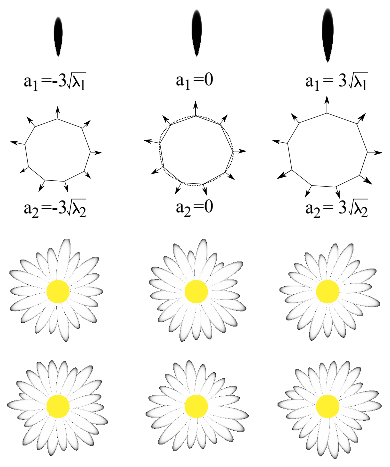
The fractal geometry of nature [Man83] suggests that many natural objects can be described as recursive compositions of similar sub-objects. So, instead of increasing the global complexity of individual figures by increasing their amount of vertices and edges, we propose to define more complex figures using compositions (unions) of n_S transformed sub-figures:

$$\Phi = \bigcup_{m=1}^{n_S} A^m \Phi^m_{(r)} \tag{2}$$

$\Phi^m_{(r)}$ is a random instance of subfigure Φ^m and A^m a corresponding transformation matrix. The subfigures may be themselves recursively defined as compositions of sub-subfigures and so forth.

In order to define random visual variations, the transformations are not deterministic but are chosen to be stochastic. Controlling statistics for defining the A^m can be achieved by using again a statistical shape model, as done for the figures. The key point is that we do not need to introduce a new specific model for the matrices A^m. We can just straightforwardly use random figure instances, as previously described, provided we are able to define a relation between both: $\Phi_{A,(r)} \rightarrow \left\{A^m_{(r)}\right\}$. $\Phi_{A,(r)}$ is a random instance of a given figure Φ_A, the latter being defined by a statistical shape model using formula 1. To define a relation between figures and transformations, we can add some more dimensions to the figure representation, i.e. each vertex $i \in [1, n_V]$ of figure Φ_A can be defined for example by five scalars instead of two: $\{v_i = (x_i, y_i, \alpha_i, sx_i, sy_i)\}$, α and (sx, sy) being respectively an angle (defining an orientation) and scales along the x and y axes. This additional information allows us to derive n_V transformations (n_V being the number of vertices of figure Φ_A) that are compositions of rotations (using the angle α_i), scalings (using sx_i, sy_i) and displacements (using x_i, y_i).

The transformation is then represented as a matrix, where the different coefficients are directly computed from the v_i. Each novel random instance of Φ_A infers a novel set of transformations A^m. By using a continuous extension of Φ_A, it is



a_1=-3$\sqrt{\lambda_1}$
a_1=0
a_1= 3$\sqrt{\lambda_1}$
a_2=-3$\sqrt{\lambda_2}$
a_2=0
a_2= 3$\sqrt{\lambda_2}$

Figure 5: *Example of two-scale statistical shape model for creating random daisy flower variations. Top: two shape models are used, (first row) for defining sub-figures (these are the petals), the other (second row) for defining jittered positions along a circle. Bottom: six random instances obtained using this model.*

furthermore possible to jitter points p_i instead of using directly its vertices v_i. Instead of fixing a constant value n_S for defining Φ (formula 2), we can choose it randomly within a minimal and maximal bound. Figure 5 illustrates a example of two-scale figure construction Φ that aims at representing a daisy flower. Φ is defined by arranging sub-figures $\Phi^m_{(r)}$ (that are petals, see first row) more or less along a circle, defined using a continuous extension of a polygon Φ_A (see second row). Using the statistical shape model, each petal might have a slightly different shape. The statistical shape model also defines variations for the transformations A^m, i.e. for the positions, orientations and scales of the petals, which is illustrated as arrows in the second row of figure 5. The dashed line in the middle represents the continuous extension of the polygon, i.e. the mean curve. The number of petals has been randomly chosen : $n_S \in [18, 25]$. Examples of random instances of daisy flowers are shown in the bottom part of figure 5. We note that we added a yellow dot, just to make the result look more like flowers.

When Φ_A is defined by a set of n_S polygons P^j, we can use these polygons to define more complex transformations A^m that are not just rotations and scalings. In fact we can make the shape of subfigure instances $\Phi^j_{(r)}$ always "fit" the shapes of the polygons P^j. To do so, we just have to express the vertices v_i of Φ^j relatively to the vertices of P^j, which can be done by using a barycentric coordinates system, similar to classical 2D texture mapping coordinates, provided the polygons are convex. Using such a coordinates system, each deformation of P^j will infer a corresponding deformation of $\Phi^j_{(r)}$. An example is shown in figure 6. The left column shows two instances of a Y-shaped figure (in red) associated to a triangle. Any transformation of this triangle infers a corresponding transformation A^m of the figure (right column).

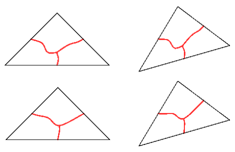


Figure 6: *Left: two random instances of a Y-shaped figure (in red) assigned to a triangle. Right: when deforming the triangle, the figure instance can be deformed accordingly by using barycentric coordinates.*

4. Procedural object distributions

In this section we show that the previous statistical figure model can be used to generate procedural basis functions consisting in defining structured random object distributions over $\Re^2$. The challenge is to make the distribution *procedural*, i.e. infinite without using repetition and computable at any location of space independently of other locations at constant time complexity. Our distribution function uses underlying periodic sets of rectilinear cells, each cell containing one figure Φ as defined by formula 1. Φ represents a geometric primitive that is used to define object positions, similarly to the way we used figures to define transformations A^m for placing sub-figures in formula 2. Since the contents of our cells are all different, the distributions are not repetitive. This is similar to jittering, where the content of each lattice square is different and computed on-the-fly.

In our case, a cell C is defined as a polygon. It may have any arbitrary shape (convex or not). The only condition is that the set of cells forms a periodic rectilinear *tessellation* of the plane, i.e. there are no gaps and no overlaps between cells. A single cell can be used. Let be $S_C = \left\{C^i\right\}$ the minimal set of cells defining the tessellation. S_C can be centered and rescaled so as to fit within one unit square. Since we assume rectilinearity, S_C represents one period of cells that can be shifted along the two unit x and y axes vectors, so as to cover entirely $\Re^2$. Each cell $C^i \in S_C$ contains a single-scale figure Φ_{C^i} as defined by formula 1. The figure's boundary may or may not directly match the shape of the cell. Parts of Φ_{C^i} may even by partly located outside the cell. Using the tessellation, it becomes straightforward to define infinite object distributions. One just has to generate for all cells of $\Re^2$, a corresponding random figure instance $\Phi_{C^i,(r)}$ and then use this instance to derive a corresponding set of positions A^m (more generally transformations, i.e. positions, scales and orientations) using the relation: $\Phi_{C^i,(r)} \rightarrow \left\{A^m_{(r)}\right\}$.

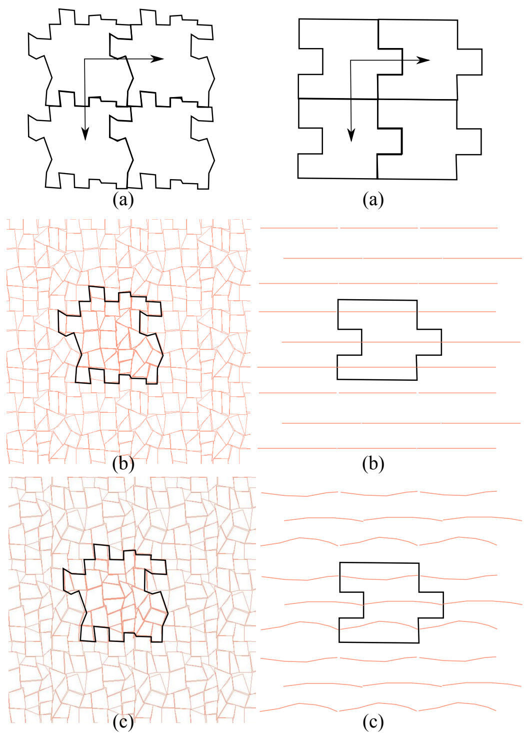


Figure 7: *Using periodic rectilinear cell-based tessellations to define object distributions. (a) a single cell representing a tessellation of the plane. (b) associating a figure Φ to the cell (in red). Φ can be composed of polygons (left) or polylines (right). (b) shows the mean shape $\bar{\Phi}$, without deformations, hence the result is periodic). Periodicity is avoided by using random instances of Φ. (c) shows the first mode, i.e. the highest deformation amplitude compared to (b).*

Figure 7 illustrates two examples of rectilinear tessellations. For each example we use a unique cell shown in part (a). The arrows represent the two unit x and y vectors. To each cell, we associate a figure composed of polygons for the left side and polylines for the right side. The mean figure $\bar{\Phi}_C$ is shown in part (b). The first mode corresponding to $3\sqrt{\lambda_1}$ is shown in part (c). The left example allows us to define object distributions using the centers of the polygons. The shape of the object may also be influenced by the shape of the polygons. For the second example, we can compute object distributions by jittering points along the curves. Because the curves are mainly horizontal, it produces object distributions characterized by strong horizontal alignments.

5. Designing and rendering procedural assemblage textures

Designing noise-like texture basis functions or procedural textures $N(x,y)$ can be done using the same mathematical formulation as for sparse convolution. The latter consists in convolving kernel functions $K(x,y)$, that have a finite spatial support, with random point distributions $p(x,y)$:

$$N(x,y) = K(x,y) * p(x,y) = \sum K(x-x_j, y-y_j)$$

where $*$ denotes the convolution operation. In our case the kernel functions are defined by multi-scale figures $K = \mathcal{R}(\Phi)$, Φ being defined by formula 2. The difference with sparse convolution is that we do not use equiprobable random point distributions (i.e. approximations of white noise). Our distribution is defined using the previously described set of cells S_C, each cell C^i being associated with a single-scale figure Φ_{C^i} used to derive positions (or more generally transformations). By drawing the set of figure samples used for statistical learning, for both the cells and the kernels, the user directly and interactively models a corresponding procedural texture.

As for sparse convolution, our assemblage technique can be directly implemented in modern GPUs at fragment level. In our case, figures are defined by sets of vertices corresponding to the mean shape, as well as sets of displacement vectors, defining the principal modes. All geometric data can a priori be stored in the form of uniform variables in the GPU shader. To compute a texture value at position (x,y), we first determine the corresponding cell C, which can be done by using the fractional parts of (x,y). Next, we compute the corresponding random figure instance of Φ_{C^i} so as to define positions, on which the kernels will be dropped. The kernels are also random instances of figures Φ as defined by formula 2. The main problem is that, as for usual sparse convolution techniques, extensively testing all positions and kernels for each rendered fragment might become excessively time consuming and break rendering performance, especially if there are many positions in one cell and/or if the spatial support of the kernels is large so that also neighboring cells must be considered. The problem is to get a fast access to the "closest" kernels that will effectively be useful for computing the convolution. Therefore, we propose an optimization that consists in reducing the number of tests by pre-computing a map where each texel contains a *stack* of potential figure candidates. We first define a map that allows us to determine which cell C is concerned (if there is a single cell such a map is not necessary). A second map, in fact one for each cell, is then used to determine which figures are concerned inside this cell. Given that each random figure instance is defined by principal modes, it allows us to compute a maximal bounding box for each figure. The map then simply precomputes indices to closest figures in the worst case (i.e. for the largest possible figure). These maps can be pre-computed by rasterizing the bounding box of each figure and storing indices directly into GPU memory using OpenGL *Image Load Store* extension mechanism. An illustration of acceleration map is shown in figure 2.

6. Results

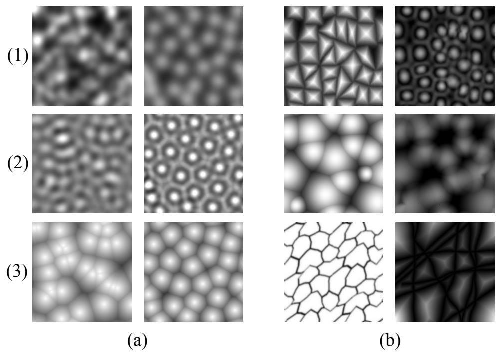


Figure 8: *Comparing classical noise patterns (a) with our new noise patterns (b). For the classical noises of (a) we used point jittering [LLDD09] (left column) and Poisson-disk distributions [LD05] (right column).*

All results in this section have been obtained with a PC using a NVidia GeForce GTX 480 for an average resolution of 1920×1200 pixels. Figure 8 shows examples of resolution independent texture basis functions. A comparison with the two available procedural random point distribution functions, i.e. jittering of points [Gla04, LLDD09] and the Poisson-disk distribution of [LD05] is shown in part (a). These two distributions are respectively used in the first and second column. They have been 1) convolved with an isotropic Gaussian kernel function, which produces Gaussian noise patterns and 2) convolved with a cardinal sinus function, which also produces noise-like patterns. In row 3), we used the closest Euclidian distance to derive cellular functions as suggested in [Wor96]. On the right part (b), we show results obtained with our approach. These texture basis functions represent new classes of structured random patterns that cannot be directly obtained with the point distributions shown in part (a). All random patterns are based on a tessellation using a single periodic cell. The first two top examples are obtained using the tessellation shown in figure 7 (left). Note that unlike the cellular texture basis function of [Wor96], users can control the shapes of the polygonal cells. The second row has been obtained using figures that correspond to stochastic triangulations of the plane. The underlying triangular structure remains just noticeable. In the last row, the left example has also been obtained using a random triangulation. In this case each triangle contains the mean Y-shape of figure 6, thus generating a flakes-like structure. Finally, the right example illustrates the case of figures composed of segments, to which we computed distances. This generates a cellular pattern that looks similar

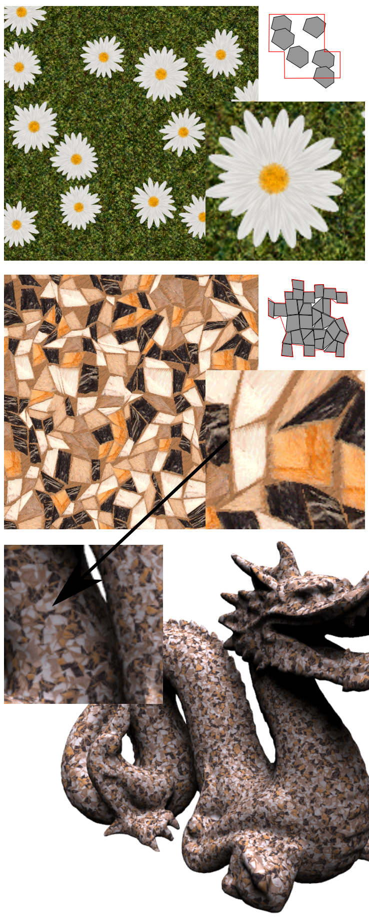


Figure 9: *Associating patterns to figures. The patterns can be defined using specific sub-figures by adding a color dimension to the vertices or by using noise-based patterns / texture images.*

to [Wor96], but with the new visual characteristic that resulting cell edges are aligned. Our texture basis functions can be directly used for texturing objects, which is shown in figure 1. The patterns are shown in the upper left corners and are applied to a 3D torus object. Instead of using distance / kernel functions to create resolution independent texture basis functions, we can also associate "patterns" to figures and sub-figures, which is similar to classical bombing [Gla04]. Two examples are illustrated in figure 9. For the top example, we used the daisy flower model defined in figure 5. The grass pattern as well as the yellow pattern in the center of the flower are defined by Gabor noises using an approach similar to [GDS10]. This way no texture memory is required. The entire texture remains totally procedural. For the second example, representing a mosaic-stone pattern, we used texture images. These texture images require less than 20Kb texture memory. The final procedural texture is non-

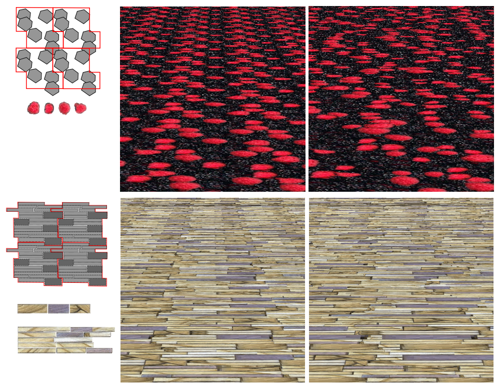


Figure 10: *Contrary to simple repetition of texture input (b), our assemblage method can be used to non-repetitively texture large surfaces (c).*

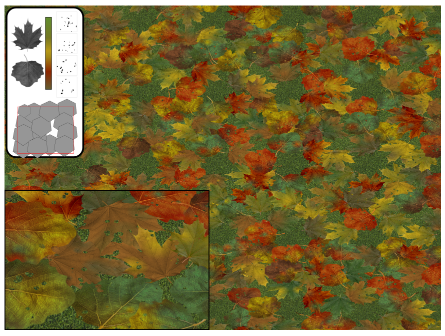


Figure 11: *By associating texture images to figures and subfigures, our assemblage technique can produce a great variety of appearances for real-time applications (70 fps).*

repetitive and quasi-infinite (see textured Dragon model). It uses the single cell model of the first column of figure 7. Figure 10 further illustrates that repetition is avoided with our approach, provided the statistical deformation model permits a sufficiently large shape variety. The middle illustrates the use of the mean figure only. Despite a random selection of patterns, periodicity appears. The right shows the use of all modes. Periodicity is no longer visible.

Finally, figures and subfigures can be associated to crops of texture photographs to produce high definition virtual textures for real-time rendering application. The leaves-covered floor shown in figure 11 shows the variety of appearances and phenomena, such as ageing, that can be represented using our technique. Here, using [PTMG08], each leaf can be rendered at real-time (70 fps) with a unique appearance by using a stochastic combination of two subfigures (with a stochastic palettized coloration)stored in GPU texture memory. Furthermore, our method can also be used to generate

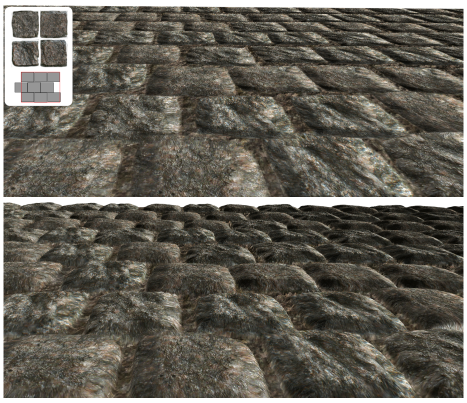


Figure 12: *Our method can also be used with displacement or vector mapping for high-quality non-repetitive 3D details. Top : 2D rendering using our assemblage technique (180 fps), bottom : Assembled texture with on-the-fly tessellation and displacement mapping (60 fps).*

on the fly high-definition non-repetitive 3D details by associating height maps or vector maps to figures.

Figure 12 shows an example of 3D details using the tessellation unit at 60 fps instead of 180 fps for simple 2D texturing (the performance hit is mostly due to the heavy tessellation of the base surface). Our assemblage method can also be used as a high-definition interactive texturing technique (40/100 fps with/without displacement mapping), as shown by the close-ups of the tree in figure 13. The texture images associated to the figures are crops of photographs stored in two 1024×1024 RGBA textures (for color and normal/height information) and can therefore be used to generate various appearances. All assemblage textures are generated in 2D space, and are thus prone to the usual stretching and discontinuity problems linked to 2D parameterizations. While discontinuities could be avoided for simple shapes by forcing textures to be periodic (i.e. well-tiling), a parameterization-aware texturing scheme should be used for more complex shapes (for instance, by extending figures across seams).

One current limitation of our method is the need to design "samples" of figures by hand for statistical learning. Multiple sample figures must be designed for object distributions and for defining visual elements (sub-figures). For complex textures, this task can be long. Another limitation is that we pre-defined a finite set of interpolation / distance functions for creating resolution independent texture basis functions. For defining other (more complex) functions it will require some programming efforts for users. Furthermore, filtering for antialiasing remains a difficult task with our method. For far-away view, a mip-mapped texture image representing a periodic distribution of figures can be used. While this is efficient in some cases, our method still lacks a complex filtering scheme to account for variations and a multi-scale definition.

7. Conclusion

Creating procedural textures is not a simple problem in computer graphics, especially when dealing with structured patterns that cannot be well represented using only noise functions. We presented a new multi-scale approach that allows users a better control of visual structures characterized by more complex dependencies among placements and kernel shapes. We used the concept of statistical shape model for defining both point distributions and kernel functions. The GPU implementation we presented allows us to texture at real-time rates very large, potentially infinite surfaces at low memory cost without repetition. As for classical bombing, complex natural textures can be defined by using textured polygons instead of kernel functions (which are rather used to define resolution independent texture basis functions).
As we have already mentioned, our approach is purely interactive and might require some efforts from users. A next step would consist in using our assemblage textures to compute procedural textures automatically from example photographs. In this case, the problem would consist in extracting "figures" from the example, these figures defining both distributions and visual elements (texture features). Another future work will consist in extending our 2D assemblage technique to the 3D case, so as to be able to edit procedural solid textures.

References

[ASK*05] Anguelov D., Srinivasan P., Koller D., Thrun S., Rodgers J., Davis J.: Scape: shape completion and animation of people. *ACM Trans. Graph. 24*, 3 (2005), 408–416. 3

[BD04] Bourque E., Dudek G.: Procedural texture matching and transformation. *Computer Graphics Forum 23*, 3 (2004), 461–468. 3

[CSHD03] Cohen M. F., Shade J., Hiller S., Deussen O.: Wang tiles for image and texture generation. *ACM Trans. Graph. 22*, 3 (2003), 287–294. 3

[DM98] Dryden I. L., Mardia K. V.: *Statistical Shape Analysis*. John Wiley & Sons, 1998. 3

[DMLG02] Dischler J.-M., Maritaud K., Lévy B., Ghazanfarpour D.: Texture particles. In *Eurographics conference proceedings* (Sep 2002). 2

[DRS08] Dorsey J., Rushmeier H., Sillion F.: *Digital Modeling of Material Appearance*. Morgan Kaufmann/Elsevier, 2008. 1

[DZ06] Dischler J.-M., Zara F.: Real-time structured texture synthesis and editing using image-mesh analogies. *Vis. Comput. 22*, 9 (Sept. 2006), 926–935. 3

[EMP*98] Ebert D., Musgrave K., Peachey P., Perlin K., Worley S.: *Texturing and Modeling: A Procedural Approach*. 1998. 1, 2

[FFC82] Fournier A., Fussell D., Carpenter L.: Computer rendering of stochastic models. *Commun. ACM 25*, 6 (June 1982), 371–384. 3

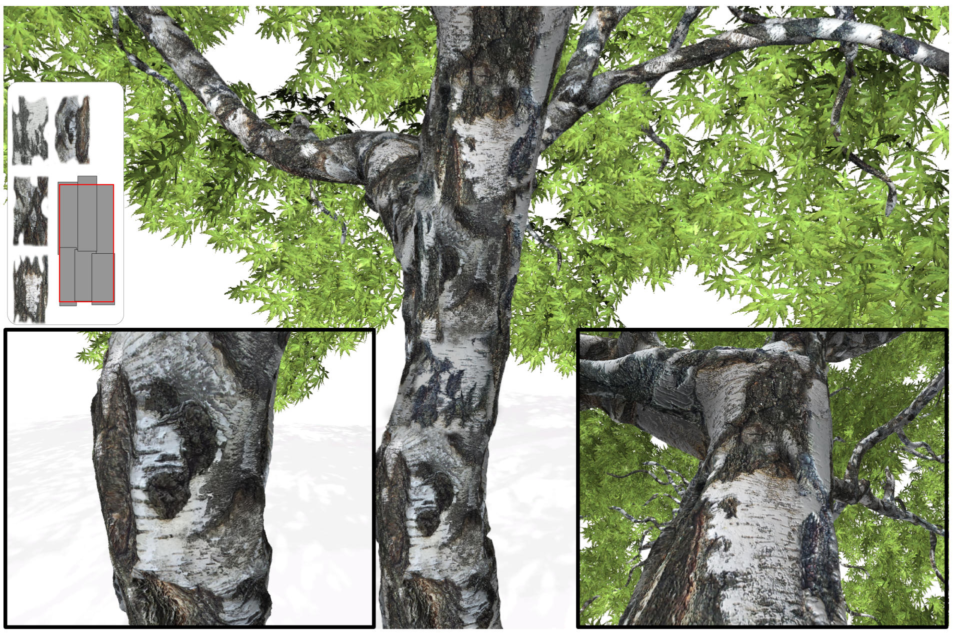


Figure 13: *All details rendered by our method can have a very-high definition (see zooms) with real-time performances (40 fps).*

[GDS10] GILET G., DISCHLER J.-M., SOLER L.: Procedural descriptions of anisotropic noisy textures by example. In *Eurographics (Short)* (2010). 3, 8

[Gla04] GLANVILLE R.: *GPU Gems*. 2004, ch. 20, Texture Bombing. 2, 3, 7, 8

[GLLD12] GALERNE B., LAGAE A., LEFEBVRE S., DRETTAKIS G.: Gabor noise by example. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012) 31*, 4 (July 2012), 73:1–73:9. To appear. 3

[LD05] LAGAE A., DUTRÉ P.: A procedural object distribution function. *ACM Transactions on Graphics 24*, 4 (October 2005), 1442–1461. 3, 7

[Lew89] LEWIS J.: Algorithms for solid noise synthesis. In *Computer Graphics ACM Siggraph annual Conference Series* (1989), pp. 263–270. 2

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: Texture sprites: Texture elements splatted on surfaces. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)* (April 2005). 2

[LLC*10] LAGAE A., LEFEBVRE S., COOK R., DEROSE T., DRETTAKIS G., EBERT D., LEWIS J., PERLIN K., ZWICKER M.: A survey of procedural noise functions. *Computer Graphics Forum 29*, 8 (December 2010), 2579–2600. 2

[LLD12] LASRAM A., LEFEBVRE S., DAMEZ C.: Procedural texture preview. *Computer Graphics Forum (Eurographics conf. proc.)* (2012). 3

[LLDD09] LAGAE A., LEFEBVRE S., DRETTAKIS G., DUTRÉ P.: Procedural noise using sparse gabor convolution. In *SIGGRAPH 2009* (2009), pp. 1–10. 2, 3, 7

[LVLD10] LAGAE A., VANGORP P., LENAERTS T., DUTRÉ P.: Procedural isotropic stochastic textures by example. *Computers & Graphics (Special issue on Procedural Methods in Computer Graphics) 34*, 4 (August 2010), 312–321. 3

[Man83] MANDELBROT B. B.: *The Fractal Geometry of Nature*. W. H. Freedman and Co., New York, 1983. 5

[MGHS11] MARVIE J.-E., GAUTRON P., HIRTZLIN P., SOURIMANT G.: Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface 2011* (2011), pp. 167–174. 3

[OHL*08] OWENS J., HOUSTON M., LUEBKE D., GREEN S., STONE J., PHILLIPS J.: Gpu computing. *Proceedings of the IEEE 96*, 5 (may 2008), 879 –899. 2

[Per85] PERLIN K.: An image synthesizer. In *Computer Graphics ACM Siggraph annual Conference Series* (1985), pp. 287–296. 2

[PTMG08] PEYRAT A., TERRAZ O., MERILLOU S., GALIN E.: Generating vast varieties of realistic leaves with parametric 2gmap l-systems. *Vis. Comput. 24*, 7 (2008), 807–816. 3, 8

[RB85] REEVES W. T., BLAU R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph. 19*, 3 (July 1985), 313–322. 2

[SA79] SCHACHTER B., AHUJA N.: Random pattern generation processes. *Computer Graphics and Image Processing 10*, 2 (1979), 95–114. 3

[vW91] VAN WIJK J. J.: Spot noise texture synthesis for data visualization. In *SIGGRAPH* (1991), pp. 309–318. 2

[WLKT09] WEI L.-Y., LEFEBVRE S., KWATRA V., TURK G.: State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report* (2009). 1, 3

[Wor96] WORLEY S.: A cellular texture basis function. In *SIGGRAPH 1996* (1996), pp. 291–294. 2, 7, 8

[ZG02] ZELINKA S., GARLAND M.: Towards real-time texture synthesis with the jump map. In *Proceedings of the 13th Eurographics workshop on Rendering* (2002), pp. 99–104. 3