Dendry: A Procedural Model for Dendritic Patterns

Mathieu Gaillard
Purdue University
mgaillar@purdue.edu

Bedrich Benes
Purdue University
bbenes@purdue.edu

Eric Guérin
Université de Lyon
eric.guerin@liris.cnrs.fr

Eric Galin
Université de Lyon
eric.galin@liris.cnrs.fr

Damien Rohmer
LIX, Ecole Polytechnique, CNRS
damien.rohmer@polytechnique.edu

Marie-Paule Cani
LIX, Ecole Polytechnique, CNRS
marie-paule.cani@polytechnique.edu

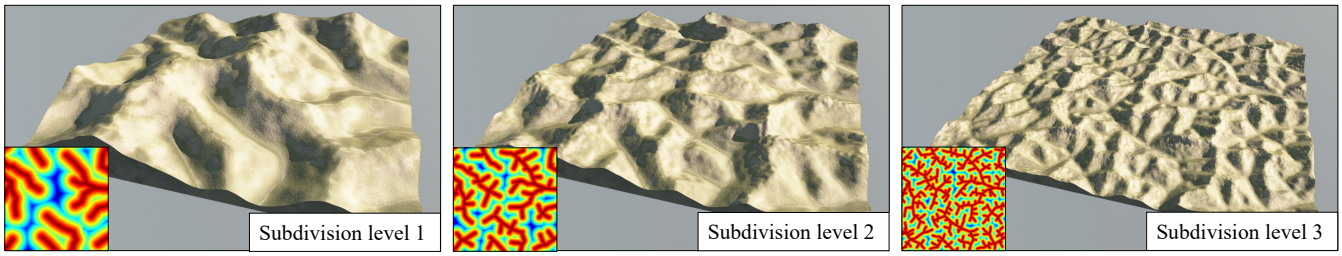


Figure 1: *Dendry* is a locally computable procedural function that generates branching patterns at various scales. One application is terrain synthesis.

ABSTRACT

We introduce *Dendry*, a procedural function that generates dendritic patterns and is locally computable. The function is controlled by parameters such as the level of branching, the degree of local smoothing, random seeding and local disturbance parameters, and the range of the branching angles. It is also controlled by a global control function that defines the overall shape and can be used, for example, to initialize local minima. The algorithm returns the distance to a tree structure which is implicitly constructed on the fly, while requiring a small memory footprint. The evaluation can be performed in parallel for multiple points and scales linearly with the number of cores. We demonstrate an application of our model to the generation of terrain heighfields with consistent river networks. A quad core implementation of our algorithm takes about ten seconds for a 512×512 resolution grid on the CPU.

CCS CONCEPTS

• **Theory of computation** → **Generating random combinatorial structures**; • **Computing methodologies** → *Shape analysis.*

KEYWORDS

Procedural Modeling, Geometric Modeling, Dendritic Patterns, Terrain Modeling

I3D '19, May 21–23, 2019, Montreal, QC, Canada

ACM ISBN 978-1-4503-6310-5/19/05…$15.00
https://doi.org/10.1145/3306131.3317020

ACM Reference Format:
Mathieu Gaillard, Bedrich Benes, Eric Guérin, Eric Galin, Damien Rohmer, and Marie-Paule Cani. 2019. Dendry: A Procedural Model for Dendritic Patterns. In *Symposium on Interactive 3D Graphics and Games (I3D '19), May 21–23, 2019, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3306131.3317020

1 INTRODUCTION

Procedural models generate large and detailed virtual content. Procedural modeling in computer graphics encompasses texture (wood, stone, marble) and geometry synthesis (terrains, buildings, facades, clouds, vegetation). Among procedural approaches, *locally computable procedural functions*, *i.e.,* functions that can be evaluated at a spatial position without requiring the knowledge of the context of the procedural function are of particular interest. Such functions can generate large models with a small memory footprint, and can be trivially evaluated in parallel. Locally computable procedural functions can be combined with other procedural functions or they can be evaluated at different scales to add details.

Previous work has shown successful examples of procedural functions such as noises [Lagae et al. 2010, 2009; Perlin 1985; Worley 1996] which are extensively used in large number of applications and industry. Noise functions are generally defined as basis functions that return a scalar value over $\mathbb{R}^n$. By construction, their output exhibits either no or only local structure patterns, which are useful for synthesizing textures or geometric models because they resemble random structures commonly found in nature.

We introduce *Dendry*; a locally computable procedural function that generates branching patterns commonly needed in computer graphics applications to model, for instance, cracks over glass or ceramic material, drying patterns on mud, snowflakes, Lichtenberg figures, or drainage systems with small creeks converging

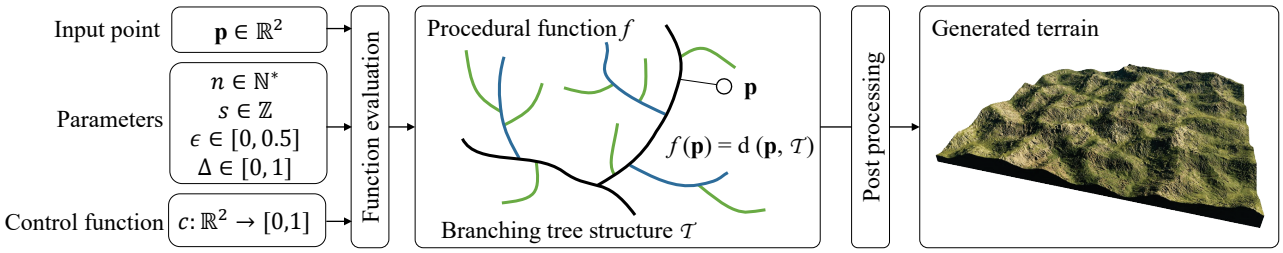


Figure 2: Overview of our method. The procedural function f is controlled by a set of parameters and by a control function c. The value of $f(\mathbf{p})$ is defined as the distance of the point $\mathbf{p}$ to the tree structure $\mathcal{T}$. It can be evaluated in parallel for multiple points, and we show how it can generate consistent terrains with drainage areas.

to large rivers. Branching patterns can be computed procedurally by using L-systems [Prusinkiewicz and Lindenmayer 1990] or by application-dependent algorithms where the branching structure is an emergent phenomenon from the simulation, such as diffusion-limited aggregation [Witten and Sander 1983] or river generation from erosion [Génevaux et al. 2013; Musgrave et al. 1989].

Previous approaches either require the simulation of a physically-based system or a context-dependent function evaluation. Our procedural scalar function generates globally consistent branching patterns while still being locally computable, on the contrary to the previous work. Our model depends on a set of intuitive parameters, including a scalar function that controls the overall shape of the branching structure. The key idea consists of seeding pseudo-random seed positions on a regular cell lattice and linking them together to generate the underlying branching structure.

We show that our algorithm scales almost linearly on 64 CPU cores and Figure 1 shows an example of generating terrains with branching ridge and river networks with an increasing level of detail. Each subdivision level adds more detail and the overall shape is inherited through the iterations.

2 RELATED WORK

The generation of branching structures has been studied for decades for particular applications. Previous works include the generation of cracks due to aging, drying or weathering processes, plant modeling, and simulation of erosion. In most of the previous work, the hierarchical branching patterns are an emergent phenomenon resulting from the simulation.

Cracks and fracture simulations have been used for the creation of breaking objects into pieces and generating branching impacts [O'Brien et al. 2002; Pfaff et al. 2014]. A procedural fracture pattern generation model was developed in [Desbenoit et al. 2005]. Fractures models are mapped onto the surfaces of objects, and are procedurally carved using a volumetric representation difference to remove material.

Ice growth using a phase field method inspired by computational physics was proposed in [Kim and Lin 2003]. The method generates branching ice structures. The simulation is controlled by a seed crystal and a freeze temperature. It can be refined by using a geometric sharpening algorithm. Glondu et al. gave a physically-based fracture model whose parameters are found using a Bayesian optimization [Glondu et al. 2012].

Plant models have distinct branching structures that have been generated by forward simulations of branching [Holton 1994], inverse modeling [Pirk et al. 2012; Šťava et al. 2014], L-systems [Prusinkiewicz et al. 1994], Diffusion Limited Aggregation [Witten and Sander 1983], or by Path Planning [Xu and Mould 2007]. Simulation methods can also generate the branching structures of plants competing for resources as described in [Honda 1971; Runions et al. 2007]. Vegetation branching patterns have also been generated interactively [Lintermann and Deussen 1999] and various approaches attempted to simulate venation patterns in leaves [Hong et al. 2005; Runions et al. 2005]. Most of these methods rely on underlying biological models based on bud growth and competition for space to mimic the branching of trees.

Terrains contain branching structures such as mountains ridges and river networks. Authors have used *noise functions* [Lagae et al. 2010] and fractal processes such as the fractional Brownian motion [Fournier et al. 1982] (fBm) to generate fractal-like landforms. Using additional noises allows for modeling the underlying structure locally *e.g.*, they behave like a function which guarantees that the noise is computed context-free at any position. This makes these algorithms attractive for real-time applications. Unfortunately, because of its lack of structuring landforms, fBm cannot generate hydrologically consistent terrains.

Kelley et al. proposed an algorithm to subdivide a drainage area to create river networks and terrains simultaneously [Kelley et al. 1988]. Prusinkiewicz et al. introduced a model derived from a fractal curve that produces a river path and adapts a mid-point displacement subdivision scheme to produce a terrain [Prusinkiewicz and Hammel 1993]. Constrained subdivisions that respect a river network's or ridge network's constraints were introduced in [Belhadj and Audibert 2005]. Procedural planets with river networks have been created in [Derzapf et al. 2011]. The generation first defines the coarse altitudes and the base river network is refined in real-time by a subdivision process.

An alternative solution to generate hydrologically consistent terrains was proposed in [Génevaux et al. 2013]. The river network is generated using a grammar-like growth algorithm which computes nodes of a graph and their corresponding altitudes. Erosion algorithms can generate rivers [Benes et al. 2006; Krištof et al. 2009; Musgrave et al. 1989] and the combination of uplift and fluvial erosion was used to generate large scale terrains with dendritic river and ridge patterns in [Cordonnier et al. 2016].

By-example terrain generation [Guérin et al. 2017; Zhou et al. 2007] and interactive editing tools [Hnaidi et al. 2010] are additional options to capture branching networks and terrains, but they usually fail to guarantee consistent water flow.

We are not aware of a method capable of creating branching drainage structures at a local scale, *i.e.,* without relying on a more global simulation process.

3 OVERVIEW

Our procedural algorithm generates a two dimensional tree structure denoted by $\mathcal{T}$ (Figure 2). The input to our algorithm is a point $\mathbf{p} \in \mathbb{R}^2$ and the output is the real number value of the procedural function: $f : \mathbb{R}^2 \rightarrow \mathbb{R}$. The value of the procedural function is the *distance* of the point $\mathbf{p}$ to the underlying branching structure $\mathcal{T}$. An important property of our approach is that the procedural function does not explicitly generate the entire structure. It builds $\mathcal{T}$ only locally and computes the Euclidean distance:

$$f(\mathbf{p}) = d(\mathbf{p}, \mathcal{T}). \tag{1}$$

An immediate benefit of this approach is that the function has a small memory footprint at the expense of being more computationally demanding. In facts, it needs to construct $\mathcal{T}$ on-the-fly at each evaluation.

Control: The function is controlled by a set of fixed parameters. The number of iterations n ($n \in [1, 6]$ in our implementation) defines the amount of branching, where higher number brings more details. The seed of the pseudo random generator s is an integer value that guarantees that the generated structure will always remain the same for a given s. The limit of the random number generator $0 \leq \varepsilon \leq 0.5$ defines the bias of the generated numbers and is further explained in Section 4. The displacement $0 \leq \Delta \leq 1$ controls the overall curvature of the generated structure by defining the maximum displacement of segments. This parameter can be either a constant or can vary at every iteration and we denote it by $\Delta_i, i = 0, 1, \ldots, n-1$. The shape of $\mathcal{T}$ is affected by a *control function* $c : \mathbb{R}^2 \rightarrow [0, 1]$, which can be thought of as an underlying environment that provides initial height-values at certain positions (Section 4.1). The control function can be either a constant, a procedural noise function, a user-defined pattern, or an interpolation of existing data such as an elevation map.

Efficiency: The evaluation at each point does not depend on other points and can be done in parallel in constant time. It is a CPU-intensive algorithm with a small memory footprint, that implies low cache misses, and does not require any synchronization.

4 PROCEDURAL FUNCTION

The function f computes the distance of a point $\mathbf{p}$ to the tree structure $\mathcal{T}$. Although $\mathcal{T}$ is not constructed on the whole domain but only locally in the neighborhood of $\mathbf{p}$, we describe the global construction first.

The tree structure $\mathcal{T}$ is constructed incrementally in n iterations (Figure 3). Initially, a few coarse branches, denoted by $\mathcal{B}_0$, are generated. At each further iteration, smaller branches $\mathcal{B}_k, k = 1, \ldots, n-1$ are iteratively added. Thus the incremental construction is defined as $\mathcal{T}_k = \mathcal{T}_{k-1} \cup \mathcal{B}_k$. Note that the higher level branches can be added to any previously generated level: a branch from $\mathcal{B}_i$ can be connected to any branch from $\mathcal{B}_k$, with $k < i$. The final tree structure is defined as the union of all branches:

$$\mathcal{T} = \bigcup_{k=0}^{n-1} \mathcal{B}_k, \tag{2}$$

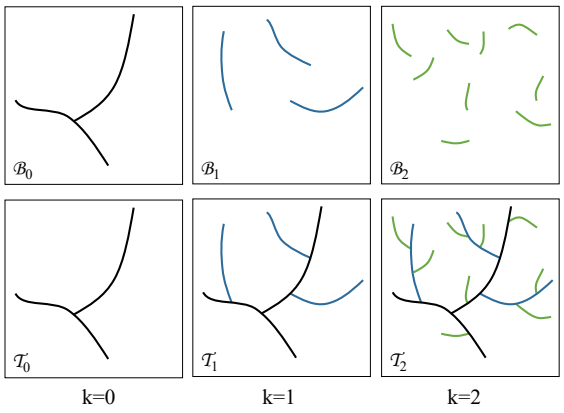


Figure 3: At each iteration k, the branches $\mathcal{B}_k$ are generated. The final tree $\mathcal{T}$ is the union of all previously generated structures.

The shape of the generated structure is affected by the control function c, but only $\mathcal{B}_0$ uses the values from the control function as explained in Section 4.1. In our experiments, using the control function at all levels over constrained the generated structure because higher levels tend to grow in the direction of the local minima of c. The control function c is also used to calculate the slope at a given point of a terrain and it is used for all levels of k (Section 5).

4.1 Point evaluation

In order to calculate the value of f at each given point $\mathbf{p}$ we need to find the closest local segment of $\mathcal{T}$. This segment is calculated only locally. At every iteration $\mathcal{B}_k$, $k \in [0, n-1]$, we overlay the entire domain by a grid $\mathcal{G}_k$, with grid cells $\mathcal{G}_k(i, j)$ (Figures 4-5). The initial resolution of the grid $\mathcal{G}_0$ is a user-defined parameter (Table 2) that defines the space between branches in $\mathcal{B}_0$.

Every grid cell $\mathcal{G}_k(i, j)$ at each level k stores a corresponding random *key-point* $\mathbf{q}_k(i, j)$ that is generated by a deterministic jittering function [Cook 1986]. Moreover, key-points from lower levels are replicated to the higher levels of resolution, so that only the key-points for the highest resolution need to be stored.

The shape of the tree $\mathcal{T}$ is affected by c. This is achieved by sampling the control function c at the locations of key-points - which, again, is done only at the lowest level $c(\mathbf{q}_0(i, j))$ (Figure 4). This value can be interpreted as color, altitude, or any other space varying scalar parameter, depending on the application, as shown in Section 5.

4.1.1 Tree initialization. In order to determine the value of $f(\mathbf{p})$ for a given input point $\mathbf{p}$, we need to find the closest point on $\mathcal{T}$. We first construct the tree $\mathcal{T}_0$ locally around $\mathbf{p}$. We find the corresponding

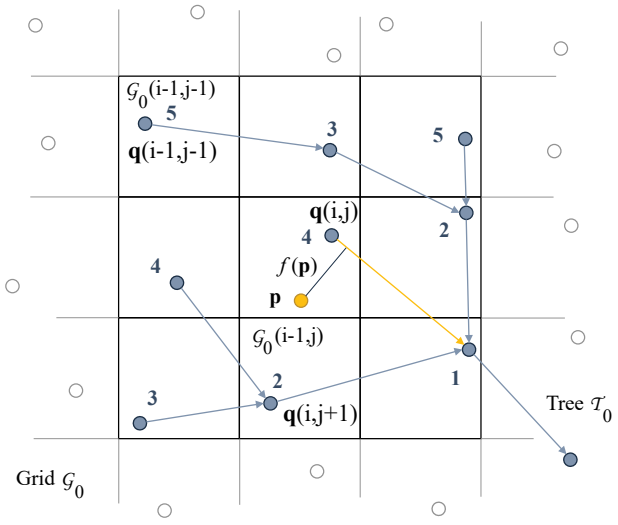


Figure 4: Generation of $\mathcal{T}_0 = \mathcal{B}_0$; only some grid cells $\mathcal{G}_0$ and points q_0 are shown for clarity. For a given point p only the local tree structure is reconstructed and the distance to the closest segment is computed and returned as $f(\mathbf{p})$.

grid cell $\mathcal{G}_0(i, j)$ for $\mathbf{p}$. Then we construct the tree by connecting all key-points in the local neighborhood (7×7 in our implementation) for $\mathcal{B}_0$. We start at the grid cell $\mathcal{G}_0(i, j)$ and connect it to the key-point with the minimal value $f(\mathbf{q}_0(i + 1, j + 1))$ in the calculated Moore neighborhood (Figure 4). This process is repeated for all key-points in the considered neighborhood. The distance to $\mathcal{T}_0$ is then the distance to the closest segment from the constructed set.

4.1.2 Generation of the higher-level branching structure. The computation of $\mathcal{T}_k$, $(k > 0)$ is performed by adding branches $\mathcal{B}_k$ that are connected to the previous tree $\mathcal{T}_{k-1}$ and to the set of new key-points $\{\mathbf{q}_k\}$. Note that while on the lowest level the value of the c function is used to determine the branch, on the higher levels we *do not* sample the control function c, instead we use the distance to the key-points.

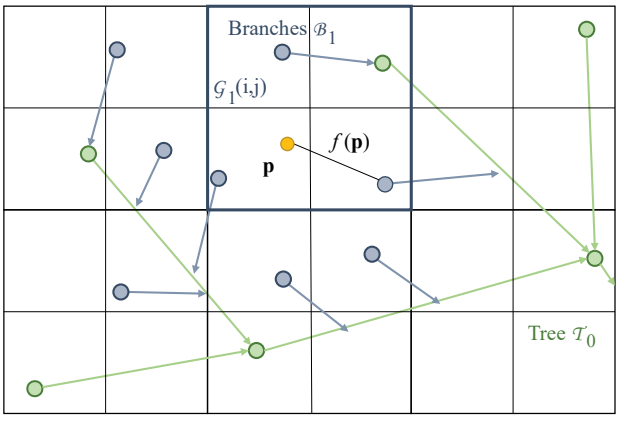


Figure 5: Generation of $\mathcal{B}_1$ on top of $\mathcal{T}_0$.

Here we describe the construction of $\mathcal{T}_1$ (Figure 5) which allows to compute the function f for $\mathcal{T}_1$. The value for higher levels is calculated in the same way from the corresponding grids. Figure 6 shows the generation of the successive levels of the underlying structure with an increasing number of iterations.

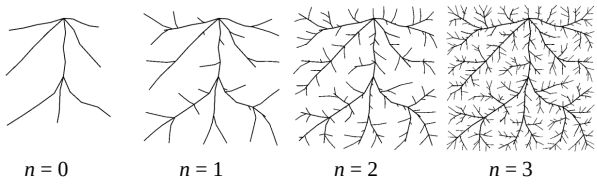


Figure 6: Increasing levels of iteration from $\mathcal{T}_0$ (left) to $\mathcal{T}_3$.

The resolution of the grid $\mathcal{G}_1$ is twice the resolution of $\mathcal{G}_0$ and reuses the previously generated key-points $\mathbf{q}_0$ of the higher level in the hierarchy. The reused points depend on their location within the cell as shown in Figure 7, where the green key-points are generated for the grid $\mathcal{G}_1$ and the white ones are inherited from $\mathcal{G}_0$.

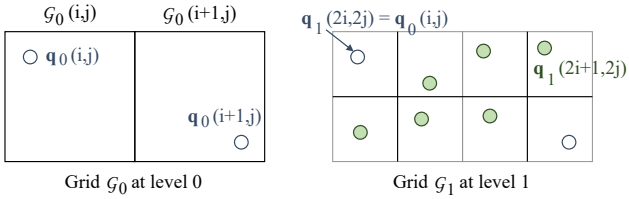


Figure 7: Point sharing between two grid levels.

The key-points are then connected to the closest segments from all previous levels ($\mathcal{T}_0, \ldots, \mathcal{T}_{k-1}$ for $\mathcal{T}_k$). Figure 5 shows that only a smaller neighborhood is required for higher levels because the structure only needs to be refined around the point $\mathbf{p}$.

At the higher levels ($k > 0$), because the jittering guarantees that points are tightly-packed, key-points connect to segments that are only up to two cells away. Hence, the evaluation neighborhood for $\mathcal{T}_k$ must be at least 5×5 cells. Because there is a variation of resolution between each consecutive level, we need to ensure that the neighborhood for all previous levels are large enough so that key-points always connect to their actual nearest segment. Let N_k be the size of the neighborhood at level k; the relationship between N_{k-1} and N_k is:

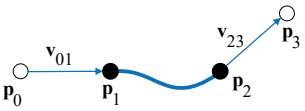
$$N_k \geq 2 \times \left\lfloor \frac{1 + N_{k-1}}{4} \right\rfloor + 3. \tag{3}$$

Compared to cellular noise and Gabor noise that require 3×3 neighborhood, we need $N_k = 5$ for all levels in our implementation. If the neighborhood was not sufficiently large, some artifacts would arise because when connecting a new point to the existing tree, its actual nearest segment would not be correctly found.

4.2 Segment smoothing and bending

The segments of the underlying structure $\mathcal{T}$ are improved by subdividing and smoothing them. We replace every segment by a cubic

spline curve defined by four control points $\{\mathbf{p}_0, \ldots, \mathbf{p}_3\}$ yielding a smooth curve from $\mathbf{p}_1$ to $\mathbf{p}_2$ whose shape is defined by the tangent vectors $\mathbf{v}_{01} = \mathbf{p}_1 - \mathbf{p}_0$ at $\mathbf{p}_1$ and $\mathbf{v}_{23} = \mathbf{p}_3 - \mathbf{p}_2$ at $\mathbf{p}_2$. Points $\mathbf{p}_1$ and $\mathbf{p}_2$ are defined as the endpoints of the segment, and are part of $\mathcal{T}$. We define the points $\mathbf{p}_0 = \mathbf{p}_1 - (\mathbf{p}_2 - \mathbf{p}1)$ so that $\mathbf{p}_1$ is the midpoint of the line segment $\mathbf{p}_0, \mathbf{p}_2$.

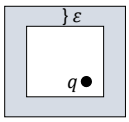


Similarly, in order to define the outgoing tangent vector $\mathbf{v}_{23}$ in $\mathbf{p}_2$, we need to find the location of $\mathbf{p}_3$. Recall that all the local segments were constructed as the closest segments (Figure 5). Therefore, the point $\mathbf{p}_3$ is defined as the end point of the segment leaving the grid cell in which $\mathbf{p}_2$ is located.

4.3 Control Parameters

The procedural function is controlled by the global parameters ε and Δ and by the control function c.

The parameter ε controls the randomness of the generated structure by constraining the distribution of the key-points $\mathbf{q}$ within the grid cells relatively to their size. Choosing $\varepsilon = 0$ allows the key-point to be generated randomly anywhere within the current grid cell $\mathcal{G}$, while an increasing value constrains the generation towards its center. Choosing $\varepsilon = 0.5$ makes the structure regular, although still affected by the control function c as shown in Figure 8.



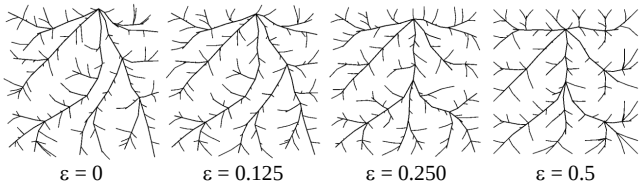


Figure 8: Constraining the randomization of the key-points $\mathbf{q}$ by increasing $\varepsilon = 0$ (left; no constraint) to $\varepsilon = 0.5$ (right; key-points are exactly in the center of each grid cell).

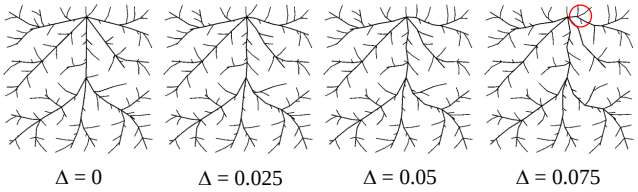


Figure 9: Increasing segment randomization.

The parameter Δ (Figure 9) increases the randomization of a segment by subdividing it into equal parts and perturbing the distance of the intermediate points to the initial segment by a random value defined as $\Delta \times l$, where l is the length of the segment. The initial segment endpoints remains unmoved. High values of Δ may cause unwanted segment intersections and yield topological changes (*e.g.*, in the structure outlined by the red circle in Figure 9). We use a maximum value of $\Delta = 0.08$.

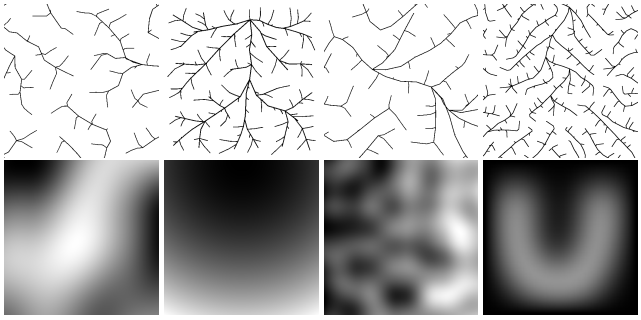


Figure 10: Control function c (bottom) significantly affects the generated structure (top). We show structures for a Perlin noise, Euclidean distance to the top middle point, Perlin noise on an inclined plane, and a user sketch.

The control function c was given as an image in our examples, but could be also generated procedurally. It is sampled during the generation of $\mathcal{B}_0$ and it is also further used for slope calculation of terrain as described in Section 5. Figure 10 shows how different control functions affect the generation of $\mathcal{T}$. Notice that the branching pattern tends to migrate to lower (darker) areas.

5 TERRAIN GENERATION

A fundamental feature of mountain ranges is the branching of ridges and river networks. By using an initial coarse height field as the control function for our procedural function, we are able to straightforwardly generate such branching structures. This method needs the minimum slope of a river segment for each level of iteration for converting the generated river network into a height-map. Figure 11 shows a terrain that was generated using a coarse sketch.

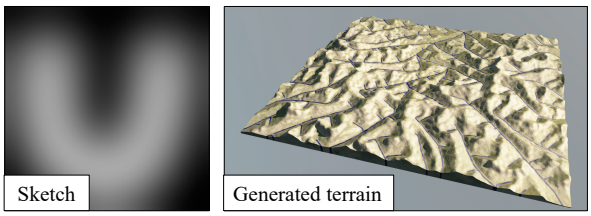


Figure 11: A U-shaped user sketch (left) and the generated terrain with enhanced rivers (right).

At the k-th iteration, we consider a grid cell $\mathcal{G}_k(i,j)$ with the corresponding key-point $\mathbf{q}_k(i,j)$. The closest segment $[\mathbf{ab}]$ to the key-point $\mathbf{q}_k(i,j)$ is also known and we want to find the point $\mathbf{q}$ on $[\mathbf{ab}]$ and to generate the subdivided and smoothed connected segment between $\mathbf{q}_k(i,j)$ and $\mathbf{q}$. The main difference in generating river segments is we should take the terrain elevation given by the control function c into account. More precisely, c should affect the bending of the river segment as seen from above to follow the direction of the gradient of the elevation.

Because river segments bend and we need some space to accommodate it, we assume that the key-point $\mathbf{q}_k(i,j)$ is connected to the nearest point among $\mathbf{a}$, $\mathbf{b}$, or to the midpoint $\mathbf{m} = (\mathbf{a} + \mathbf{b})/2$. Note

that by construction from the previous iteration, the segment **ab** has a gradient so that $h(\mathbf{a}) \leq h(\mathbf{m}) \leq h(\mathbf{b})$. After computing the connecting point $\mathbf{x} \in \{\mathbf{a}, \mathbf{m}, \mathbf{b}\}$, we set the altitude of the key-point as $h(\mathbf{q}_k(i,j)) = c(\mathbf{q}_k(i,j))$. We also know that $\mathbf{q}_k(i,j)$ must be located at a higher altitude than the connecting point $\mathbf{x}$. Therefore, we compute a minimum slope with the simplified Flint's equation from [Flint 1974] and [Kelley et al. 1988]. $S = \rho(2\mu - 1)^{\beta}$, where S is the slope of a river of magnitude μ (Shreve stream order), ρ is the average length of a tributary, and β is a negative exponent equal to -0.6. Because our algorithm is greedy, ρ is constant and μ depends only on the resolution level k of a river segment. In fact, we cannot predict in advance the length of the tributaries, as they are computed during the next iterations; $\mu = 1$ at the highest resolution level and is multiplied by three at each subsequent level. We can then bend the generated spline in the direction of the flow (Section 4.2). The actual connecting angle α of the tributary joins is computed as in [Howard 1971], using $cos(\alpha) = S_m/S_n$, where S_m is the slope of the main river and S_n is the slope of the added segment.

The tree $\mathcal{T}$ does not span the entire domain of definition so we need to assign heights to areas where the tree $\mathcal{T}$ is not defined *i.e.,* zones around rivers where $f(\mathbf{p}) > 0$. Inspired from [Génevaux et al. 2013], we compute constant height primitives on the terrain at a higher resolution than the tree $\mathcal{T}$ and average their contributions weighted by the inverse distance to point $\mathbf{p}$.

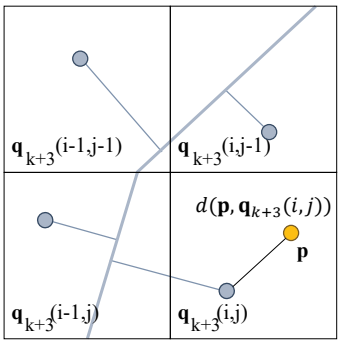


Figure 12: Calculation of $h(\mathbf{p})$ from the height primitives centered in points $\mathbf{q}_{k+3}$.

To generate the higher resolution set of points around which the constant height primitives are centered, we subdivide the grid $\mathcal{G}_k$ three more times, thus centering each primitive at a point $\mathbf{q}_{k+3}(i,j)$. See Figure 12. We then assign the elevation to these:

$$h(\mathbf{q}_{k+3}(i,j)) = h_{\mathcal{T}_k}(\mathbf{q}_{k+3}(i,j)) + \tan\theta \; f(\mathbf{q}_{k+3}(i,j)), \tag{4}$$

where $h_{\mathcal{T}_k}(\mathbf{q}_{k+3}(i,j))$ is the altitude of the nearest point on $\mathcal{T}_k$ to $\mathbf{q}_{k+3}(i,j)$. In other words, each primitive centered at point $\mathbf{q}_{k+3}(i,j)$ is placed on a θ degree slope around rivers. We merge all heights from the primitives centered at points $\mathbf{q}_{k+3}$ using the mean:

$$\bar{h}(\mathbf{p}) = \frac{\sum \alpha(i,j)\, h(\mathbf{q}_{k+3}(i,j))}{\sum \alpha(i,j)} \tag{5}$$

weighted by a smoothly decreasing function g of the distance between $\mathbf{p}$ and the points $\mathbf{q}_{k+3}(i,j)$.

$$\alpha(i,j) = g \circ d(\mathbf{p}, \mathbf{q}_{k+3}(i,j)/2^{k+2}) \qquad g(r) = \left(1 - r^2\right)^3 \tag{6}$$

In our implementation, we clamp the neighborhood to the 5×5 nearest primitives to $\mathbf{p}$ because farther distances have insignificant influence.

The slope $\tan\theta$ around rivers depends on $h_{\mathcal{T}_k}(\mathbf{q}_{k+3}(i,j))$, the height of the nearest point on the tree $\mathcal{T}$. It makes the landscape more interesting. Valleys are wider near to the global minimum of the control function and narrower near to the global maximum of the control function. Let h_{min} and h_{max} be respectively the minimum and maximum of the control function. The angle θ is calculated as a smooth-step sigmoid-shaped function:

$$\tan\theta = \sigma(h_{min}, h_{max}, h_{\mathcal{T}_k}(\mathbf{q}_{k+3}(i,j))^{\beta}). \tag{7}$$

We introduce a new parameter affecting the slope $0 \leq \beta \leq 2$ (Table 2 and Figure 13). With $\beta \approx 0$ the slopes are uniform, and increasing β adds more valleys. A high value ($\beta > 1$) creates sharp ridges similar to cliffs. We found that high values ($\beta > 1$) are more suited to small terrains whereas small values better captures large terrains (Table 2 reports the values used for the different figures in the paper). Figure 13 shows the effect of increasing β on the terrain.

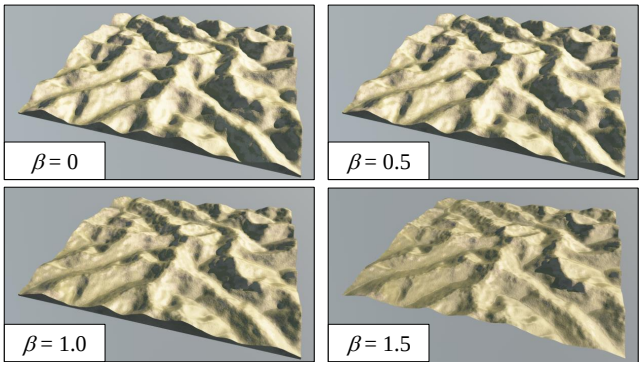


Figure 13: The effect of β on the generated terrain.

6 RESULTS

We implemented our algorithm in C++ and we generated results on a laptop computer equipped with an Intel i5-6300U clocked at 2.90GHz and 16GB of DDR3 RAM running Windows 10. Moreover, we tested the parallelization on a system equipped with an Intel Xeon Phi 7210 processor with 64 cores (hyper-threaded to 256) at 1.30GHz (Turbo Frequency 1.50GHz), 32 MB L2 cache, 16GB of MCDRAM, 96GB of DDR4-2133 on 6 channels, running CentOS 7.3. The performance and the parameters for all results in this paper are reported in Table 2.

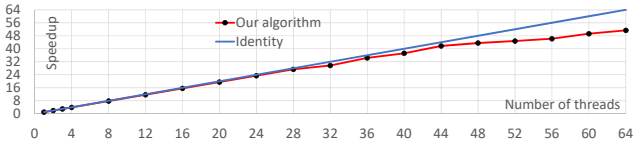


Figure 14: Our parallel algorithm scales almost linearly with increasing number of cores.

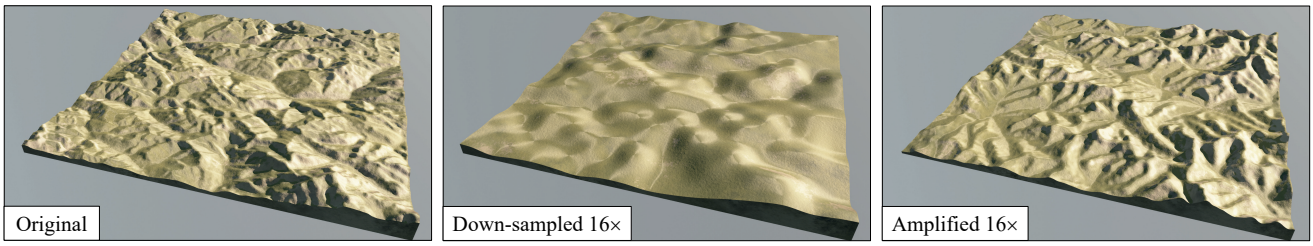


Figure 15: An example of a real DEM terrain $8 \times 8\,\mathrm{km}^2$ from Alps (left) that has been down-sampled (middle) and used as control function c and amplified to the original resolution by our algorithm (right).

Figure 14 shows the **scalability** of our algorithm. We executed the program on Intel Xeon Phi on $1, 2, \ldots, 64$ cores, and compared the measured speedup to the ideal one. The speedup in Figure 14 is reported by forcing each core to run exactly one instance of the algorithm. The speedup for 64 cores was 51.4. The Xeon Phi architecture is capable of hyper-threading of up to four threads per core, but there is a performance hit per thread. By using hyper-threading the speedup for 256 threads was 106.6.

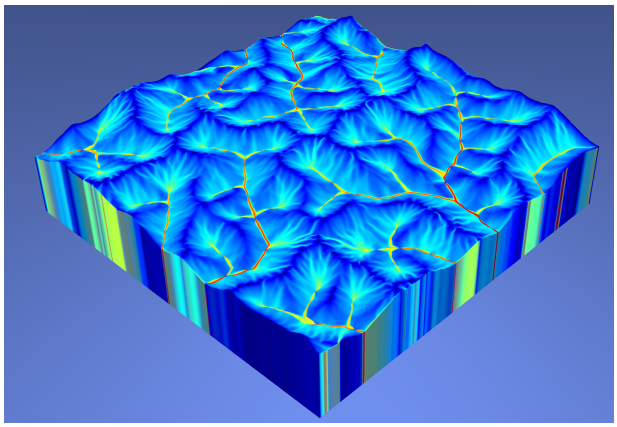


Figure 16: The terrain from the center of Figure 1 with a texture showing the logarithm of the drainage area in every cell.

A *drainage network* generated by our algorithm is shown in Figure 16. By construction, our algorithm guarantees that water flows towards the borders of the terrain, provided that the control function c does not include too low local minima. Every river segment has a consistent slope in the direction of the root of the tree $\mathcal{T}$. The control function has a high visual impact on the resulting river network. If it is convex, it will avoid depression formation. Our method to transform the river network into a heightmap may introduce some high frequency noise, which is why there are still some local minima. While we could generate hydrology consistent graph without any local minima and thus depressions, this would over-constrain the elevation and result into less natural terrains as slopes around rivers would then be perfectly flat and ridges would not be continuous.

We compare the hydrological consistency of our method to other procedural generation algorithms. The flow of water is simulated on the terrain as if it were an impervious surface: any depression in the terrain is filled with water. The more depressions, the worse the drainage is. Depressions are computed using the Priority-flood algorithm from [Barnes et al. 2014]. We use a D-Infinity Flow algorithm to compute the drainage area in every cell of the terrain as shown in Figure 16. To evaluate the different methods, we compute the number and relative surface area of depressions (we measured the ratio of surfaces because most of the procedurally generated terrains we compare to are dimensionless). We compared our method to other procedural functions (Figure 18). For our method, we averaged 10 different terrains generated from 10 different seeds. For reference, we evaluated the statistics for a real $30 \times 30\,\mathrm{km}^2$ DEM in the Alps, Europe. Results can be found in Table 1 and show that our method generates significantly less depressions than other procedural methods which produce many local minima.

Table 1: Hydrology statistics.

Method	Depression surface	# of depressions
Multifractal Fig 17	35.9 %	1352
Ridged Fig 17	24.5 %	2058
Ours (average of 10 seeds)	0.8 %	52
Alps	3.3 %	798

Amplification: Figure 15 shows a real digital elevation model (DEM) of a terrain from Alp $7.7 \times 7.7\,\mathrm{km}^2$ at 256×256 resolution corresponding to approximately 30 m per cell. The same terrain has been down-sampled to 16×16 resolution (middle) and amplified by our algorithm. Our algorithm adds more details than were present in the original while preserving overall appearance.

Comparison: We compared *Dendry* to methods that generate details on an initial coarse terrain, in particular hydraulic erosion and sparse amplification [Guérin et al. 2016] (Figure 17). Our function has the property of being fully local. Hydraulic erosion allows the carving of channels that guarantee consistent water flow, but at a high computational cost (several minutes in this example). Sparse amplification allows generation of realistic details that are provided by a real-data source, but lacks of global coherency especially in terms of flowing consistency. In comparison, our method adds branching channels to the initial terrain while preserving the consistency of water flow and is more effective since it can be evaluated locally.

Table 2: Statistics for different images show throughout the paper: model, terrain resolution, grid size, random number generator seed, number of sub-trees n, randomness factor ε.

Model	Resolution	Grid	Seed	n	ε	Δ	c	Slope Power β	Runtime (s)
Fig 1 left	512×512	4×4	0	1	0.25	0.075	Perlin Noise	0.5	47
Fig 1 center	768×768	4×4	0	2	0.25	0.075	Perlin Noise	0.5	96
Fig 1 right	1024×1024	4×4	0	3	0.25	0.075	Perlin Noise	0.1	269
Fig 2	768×768	4×4	0	2	0.25	0.075	Perlin Noise	0.5	96
Fig 6	512×512	3×3	5	1 - 4	0.25	0.05	Radial ramp	N.A.	12, 24, 38, 46
Fig 8	512×512	3×3	5	3	0 - 0.5	0	Radial ramp	N.A.	38
Fig 9	512×512	3×3	5	3	0.25	0 - 0.075	Radial ramp	N.A.	38
Fig 13	512×512	4×4	1	2	0.25	0.08	Perlin Noise	0 - 1.5	31
Fig 11	1024×1024	8×8	0	2	0.15	0.025	Sketch	0.5	216
Fig 16	768×768	4×4	0	2	0.25	0.075	Perlin Noise	0.5	96
Fig 15	256×256	16×16	0	1	0.10	0.05	Real Terrain (Alps)	0.75	25
Fig 17	512×512	4×4	0	2	0.25	0.08	Real Terrain (Alps)	1.5	64
Fig 18	1024×1024	4×4	0	3	0.25	0.075	Perlin Noise	0.1	269

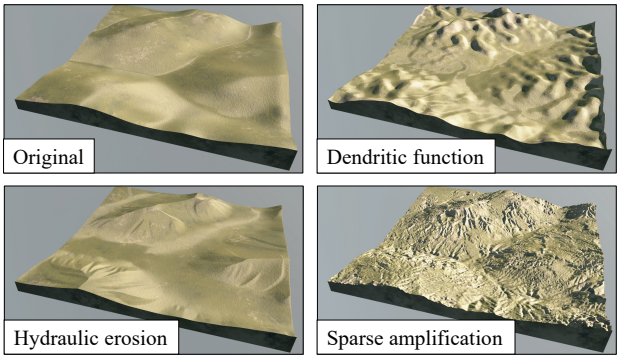


Figure 17: Comparison of synthesized details produced by different amplification methods: the original terrain (top left) had a coarse resolution of 32×32. We compare our dendritic function (top-right) to hydraulic erosion (bottom-left) and sparse amplification (bottom-right), at 512×512 resolution, *i.e.,* an amplification factor of 16.

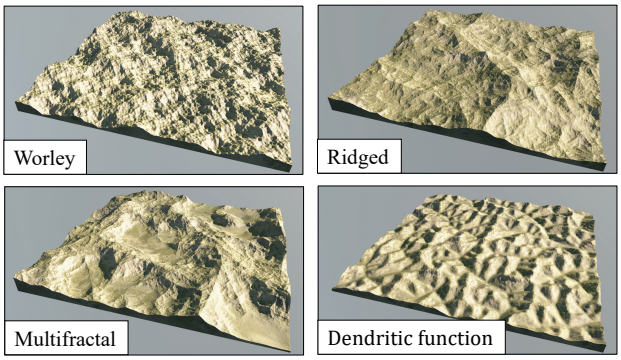


Figure 18: Comparison of our dendritic function (right) with different noise functions to generate synthetic terrains: cellular noise (left), ridged noise (center-left), multi-fractal noise (center-right).

We also compared our dendritic function to other procedural functions that are locally computable (Figure 18). Traditional noises provide locally realistic landform features such as crests but fail to generate a coherent drainage.

7 CONCLUSIONS AND FUTURE WORK

We have introduced a novel procedural function that is controlled by a few intuitive parameters which returns a distance to an underlying branching structure and is evaluated locally. We demonstrated its application to the generation of terrains with river networks.

While our method is straightforward and lends itself for parallel implementation, it has several *limitations*. First the regular grid causes a fixed density of branches. They could be avoided by using a non-regular grid. Targeting a small memory footprint also comes with its counterpart: many local branches are repeatedly calculated. It would be possible to increase the memory footprint by caching some of the already calculated structures. One strategy would be to use lazy evaluation and storing the intermediate results in each cell. Another limitation is the dependence of the random parameters on the size of the grid. Unwanted crossing between segments can occur in the generated tree structure when values of parameters are too high. As explained in Section 4.3, it happens mostly with Δ and ϵ, and it is slightly amplified when segments are smoothed.

There are several possible extensions to our work. We could apply the function to other examples. In particular, we think about Lichtenberg figures because the density of branches is almost constant. It would be interesting to see how the function could return two or, in general, multiple closest points to $\mathcal{T}$ in order to compute more complex functions. Using non-regular grids would probably alleviate the problem of having a constant density of branches. Two other ways of mitigating this effect could be: modifying the slope around segments according to a slope map or combining this procedural function with other functions that are better at modeling different reliefs. Although we did not implement a non-local version on the CPU, it could be interesting to compare its run time with the

local version. Another straightforward extension is the implementation on the GPU. Because of the local property of the model and its grid-based computation, it could be implemented as a shader. Finally, interesting patterns could be generated by extending our algorithm to 3D, at the expense of more demanding computations.

ACKNOWLEDGMENTS

We would like to thank Intel for providing a Xeon Phi computer. This research was funded in part by National Science Foundation grants #10001387, *Functional Proceduralization of 3D Geometric Models* and grant #1608762, *Inverse Procedural Material Modeling for Battery Design*. This work was also supported by the project *PAPAYA* P110720-2659260, funded by the Fonds National pour la Société Numérique and the project *HDW* ANR-16-CE33-0001.

REFERENCES

Richard Barnes, Clarence Lehman, and David Mulla. 2014. Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers and Geosciences* 62 (2014), 117–127. arXiv:1511.04463

Farès Belhadj and Pierre Audibert. 2005. Modeling Landscapes with Ridges and Rivers: Bottom Up Approach. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. ACM, Dunedin, New Zealand, 447–450.

Bedrich Benes, Václav Těšínský, Jan Hornyš, and Sanjiv K. Bhatia. 2006. Hydraulic erosion: Research Articles. *Comput. Animat. Virtual Worlds* 17, 2 (2006), 99–108.

Robert L. Cook. 1986. Stochastic Sampling in Computer Graphics. *ACM Trans. Graph.* 5, 1 (Jan. 1986), 51–72.

Guillaume Cordonnier, Jean Braun, Marie-Paule Cani, Bedrich Benes, Eric Galin, Adrien Peytavie, and Eric Guérin. 2016. Large Scale Terrain Generation from Tectonic Uplift and Fluvial Erosion. *Computer Graphics Forum* 35, 2 (May 2016), 165–175.

Evgenij Derzapf, Björn Ganster, Michael Guthe, and Reinhard Klein. 2011. River Networks for Instant Procedural Planets. *Proceedings of Pacific Graphics 2011* (2011), 2031–2040.

Brett Desbenoit, Eric Galin, and Samir Akkouche. 2005. Modeling cracks and fractures. *The Visual Computer* 21, 8-10 (2005), 717–726.

Jean-Jacques Flint. 1974. Stream gradient as a function of order, magnitude, and discharge. *Water Resources Research* 10, 5 (1974), 969–973.

Alain Fournier, Don Fussell, and Loren Carpenter. 1982. Computer Rendering of Stochastic Models. *Commun. ACM* 25, 6 (June 1982), 371–384.

Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. 2013. Terrain Generation Using Procedural Models Based on Hydrology. *ACM Trans. Graph.* 32, 4, Article 143 (July 2013), 143:1–143:13 pages.

Loeiz Glondu, Lien Muguercia, Maud Marchal, Carles Bosch Geli, Holly Rushmeier, George Dumont, and George Drettakis. 2012. Example-Based Fractured Appearance. *Comp. Graph. Forum* 31, 4 (June 2012), 1547–1556.

Eric Guérin, Julie Digne, Eric Galin, and Adrien Peytavie. 2016. Sparse representation of terrains for procedural modeling. *Computer Graphics Forum (proceedings of Eurographics 2016)* 35, 2 (2016), 177–187.

Eric Guérin, Julie Digne, Eric Galin, Adrien Peytavie, Christian Wolf, Bedrich Benes, and Benoit Martinez. 2017. Interactive Example-Based Terrain Authoring with Conditional Generative Adversarial Networks. *ACM Transactions on Graphics (proceedings of Siggraph Asia 2017)* 36, 6 (2017), 1–13.

Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. 2010. Feature based terrain generation using diffusion equation. *Computer Graphics Forum (Proceedings of Pacific Graphics)* 29, 7 (2010), 2179–2186.

Matthew Holton. 1994. Strands, Gravity and Botanical Tree Imagery. *Computer Graphics Forum* 13(I) (1994), 57–67.

Hisao Honda. 1971. Description of the form of trees by the parameters of the tree-like body: effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology* 31 (1971), 331–338.

Sung Min. Hong, Bruce Simpson, and Gladmir V.G. Baranoski. 2005. Interactive venation-based leaf shape modeling. *Computer Animation and Virtual Worlds* 16 (2005), 415–427.

Alan D. Howard. 1971. Optimal Angles of Stream Junction: Geometric, Stability to Capture, and Minimum Power Criteria. *Water Resources Research* 7, 4 (1971), 863–873.

Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. 1988. Terrain simulation using a model of stream erosion. *ACM Transactions on Graphics* (1988), 263–268.

Theodore Kim and Ming C. Lin. 2003. Visual simulation of ice crystal growth. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA 2003*. 86–97.

Peter Krištof, Bedrich Benes, Jaroslav Křivánek, and Ondřej Šťava. 2009. Hydraulic Erosion Using Smoothed Particle Hydrodynamics. *Computer Graphics Forum (Proceedings of Eurographics 2009)* 28, 2 (mar 2009), 219–228.

Ares. Lagae, Sylvain Lefebvre, Rob Cook, Tony. DeRose, George Drettakis, David S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. 2010. A survey of procedural noise functions. *Computer Graphics Forum* 29, 8 (2010), 2579–2600.

Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. 2009. Procedural Noise using Sparse Gabor Convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)* 28, 3 (July 2009), 54–64.

Bernd Lintermann and Oliver Deussen. 1999. Interactive Modeling of Plants. *IEEE Comput. Graph. Appl.* 19, 1 (1999), 56–65.

F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. 1989. The synthesis and rendering of eroded fractal terrains. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. ACM Press, New York, NY, USA, 41–50.

James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins. 2002. Graphical Modeling and Animation of Ductile Fracture. In *Proceedings of ACM SIGGRAPH 2002*. ACM Press, 291–294.

Ken Perlin. 1985. An Image Synthesizer. *SIGGRAPH Comput. Graph.* 19, 3 (July 1985), 287–296.

Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. 2014. Adaptive Tearing and Cracking of Thin Sheets. *ACM Transactions on Graphics* 33, 4 (2014), 1–9.

Sören Pirk, Ondrej Stava, Julian Kratt, Michel Abdul Massih Said, Boris Neubert, Radomír Měch, Bedrich Benes, and Oliver Deussen. 2012. Plastic Trees: Interactive Self-adapting Botanical Tree Models. *ACM Transactions on Graphics* 31, 4, Article 50 (July 2012), 50:1–50:10 pages.

Przemyslaw Prusinkiewicz and Mark Hammel. 1993. A Fractal Model of Mountains with Rivers. In *Proceedings of Graphics Interface'93*, Vol. 30(4). 174–180.

Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. 1994. Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM Press, New York, NY, USA, 351–358.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1990. *The Algorithmic Beauty of Plants*. Springer–Verlag, New York. With J.S.Hanan, F.D. Fracchia, D.R.Fowler, M.J.de Boer, and L.Mercer.

Adam Runions, Martin Fuhrer, Brendan Lane, Pavol Federl, Anne-Gaëlle Rolland-Lagan, and Przemyslaw Prusinkiewicz. 2005. Modeling and visualization of leaf venation patterns. *ACM Trans. Graph.* 24, 3 (2005), 702–711.

Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. 2007. Modeling Trees with a Space Colonization Algorithm. In *Proceedings of Eurographics Workshop on Natural Phenomena*. Eurographics Association, 63–70.

Ondřej Šťava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomír Měch, Oliver Deussen, and Bedrich Benes. 2014. Inverse Procedural Modelling of Trees. *Computer Graphics Forum* 33, 6 (2014), 118–131.

Thomas A. Witten and Leonard M. Sander. 1983. Diffusion-limited aggregation. *Phys. Rev. B* 27 (May 1983), 5686–5697. Issue 9.

Steven Worley. 1996. A Cellular Texture Basis Function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 291–294.

Ling Xu and David Mould. 2007. Modeling dendritic shapes. *Grapp 2007: Proceedings of the Second International Conference on Computer Graphics Theory and Applications, Vol Gm/R* (2007).

Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. 2007. Terrain Synthesis from Digital Elevation Models. *IEEE Transactions on Visualization and Computer Graphics* 13, 4 (2007), 834–848.