Eurographics Symposium on Rendering 2020
C. Dachsbacher and M. Pharr
(Guest Editors)

Volume 39 (2020), Number 4

Semi-Procedural Textures Using Point Process Texture Basis Functions

P. Guehl[1], R. Allègre[1], J.-M. Dischler[1], B. Benes[2], and E. Galin[3]

[1]ICube, Université de Strasbourg, CNRS, France [2]Purdue University, USA [3]LIRIS, Université de Lyon, CNRS, France

Supplemental material #4

In this supplemental material, we describe PPTBF implementation details and practical choices.

1. Window Function

Figure 1 shows an example of the influence of the cellular window w_1 parameters λ (anisotropy) and l_c (smoothness) for two different values n_v (number of Bezier curve control points).

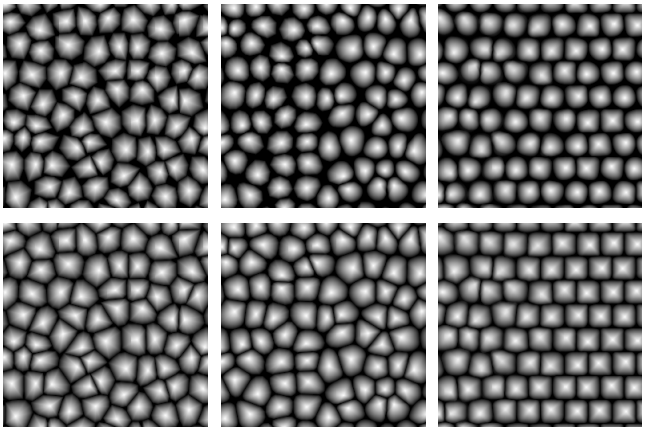

Figure 1: *Examples of the cellular window basis w_1 obtained for an increasing λ (from left to right) and by using tiling type (d). The second row uses the same parameters but a higher n_v.*

2. Feature Function

The feature function is defined by a mixture of stringed Gabor kernels $\tilde{G}_j$. Since we only want to produce rough stochastic structures, we implemented a limited number of mixtures. We experimentally and empirically found these mixtures to be able to surprisingly well cover most of the natural stochastic structures present in our database. Label $\nu_{\tilde{G}}$ for f selects the mixture among a finite set (we implemented 5 mixture models, as described next).

In parallel, we implemented simple PDFs for randomly drawing all other parameters of the feature function: J, $\mu_{\mathbf{j}|\mathbf{i}}$, $\sigma_{j|i}$, θ_j and A_j, the remaining parameters κ, ϕ and τ being constant for all $\tilde{G}_j$. As mentioned in the paper $\mu_{\mathbf{j}|\mathbf{i}}$ and $\sigma_{j|i}$ are correlated to the tessellation cells R_i. J is uniformly drawn in interval $[J_{min}, J_{max}]$. θ_j is uniformly drawn in $[0, \theta]$. We implemented only two different ways for selecting the weights A_j: 1) $A_j = 1$ for all j and 2) a random binary selection in $-1, 1$ (actually we take uniform random values "close" to -1 and 1).

The mixture model is managed by function ω_j. It is a function of position $\mathbf{x} \in \mathbb{R}^2$ that determines how kernel j will be contributing to the sum. If its value is 0 for a given kernel at a given location $\mathbf{x}$, then kernel $\tilde{G}_j$ does not contribute at all to the sum on this location. This allows us to define for example stacks of kernels instead of blends, feature stacks and piles being frequent in some natural patterns such as pebbles or foliage.

In practice, we implemented following five mixture models:

- $\omega_j = 1$ for all $\mathbf{x}$. A_j is selected in $-1, 1$. This results in a classical sum;
- $\omega_j = 1/J$ for all $\mathbf{x}$. $A_j = 1$. This is also a classical sum, but with no negative values. Therefore it is normalized;
- $\omega_j = \frac{1}{J\tilde{G}_j} - 1$. $A_j = 1$. Same sum as previous, but the result is eventually inverted. It is equilavent to computing $f(\mathbf{x}) = 1 - \sum_{j=1}^{J} \tilde{G}_j(\mathbf{x})$. As a result, features are "carved" into the window function when applying the multiplication with the latter;
- ω_j is a Kronecker delta function based on a max operator, consisting in keeping only the highest value of all stringed Gabor kernels. This should not be confused with Worley's min operator on distances, though in some case we obtain visually very similar results (depending on the other parameters of $\tilde{G}_j$.
- ω_j is a Kronecker delta function based on a max operator. This time, the maximum is computed for random values drawn on $\mu_{\mathbf{j}|\mathbf{i}}$ (the positions of the kernels). It results in stacking stringed Ga-

bor kernels in overlapping regions according to a certain random priority. This is a kind of bombing of stringed Gabor kernels.

Figure 2 illustrates these five mixture models (columns). The rows show variations of correlation parameter *correl*. In all cases we used tesselation (a) and $J_{min} = 5, J_{max} = 5$, a medium jittering value and a slight anisotropy with orientation interval $\theta_j \in [0, \pi/2]$. Frequency was set to 0, so there are no Gabor stripes.

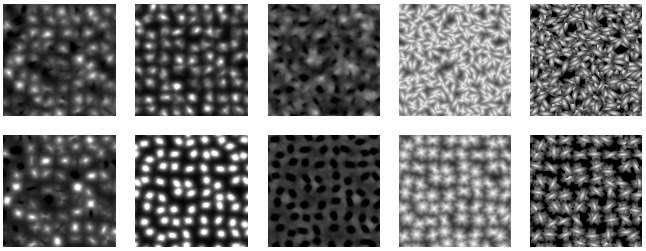

Figure 2: *Examples of PPTBF. Each column shows one of the five mixture models we implemented. The first row shows no correlation with the window centroid, the second one, a strong correlation. Being able to easily manage correlations is one of the key properties of our model. As can be observed, it has a strong influence on the visual result.*

3. PPTBF parameters and normalization

Table 1 summarizes all PPTBF parameters.

Symb.	Description	Value
POINT PROCESS (Paper Section 3.1.1.)		
ν_T	tiling type (*e.g.,* regular, irregular...)	{0,..., 16}
jit	jittering (randomness)	[0, 1]
WINDOW FUNCTION (Paper Section 3.1.2.)		
ω	linear combination of the two basis windows	[0,..., 1]
$\|\cdot\|_c$	cellular basis window norm	[1,2,..., ∞]
λ	anisotropy of cellular basis window	[0, 1]
n_v	number of control points for smoothing	[3,..., 64]
l_c	degree of smoothing	[0,..., 2]
σ_j	sigma of window	float
FEATURE FUNCTION (Paper Section 3.1.3.)		
f_t	mixture model	[0,..., 3]
J_{min} J_{max}	min/max number of kernels	[0,..., 8]
correl	correlation with centroids	[0,..., 1]
ϕ	frequency of stringed Gabor stripes	[0,..., 16]
τ	thickness of stringed Gabor stripes	[0,..., 1]
κ	curvature of stringed Gabor stripes	[0,..., 1]
θ	orientation of stringed Gabor stripes	[0,..., π/2]
η	anisotropy	[0,..., 5]
$\|\cdot\|_f$	feature norm	[1,2,..., ∞]
σ_f	sigma of Gabor kernel	float
σ_{fvar}	σ_f random variation	float

Table 1: *Overview of the control parameters.*

Visual structures are defined by thresholding the PPTBF (Fig. 3). Normalization must be applied to generate the database used to accelerate the estimation of best matching PPTBF parameters. Figure 4 illustrates the four types of deformation we apply to all images for normalization. Of course we apply multiple scales, rotations, stretches and Brownian motion-based distortions.

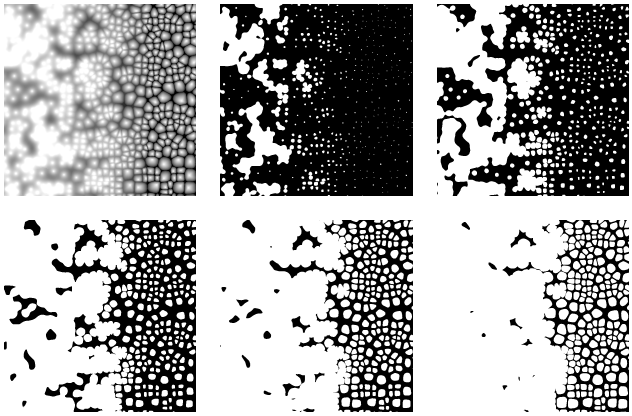

Figure 3: *Thresholding the PPTBF. A single PPTBF incorporates multiple binary structures with different topologies, that are only revealed by different thresholds. Top-left: PPTBF as a grayscale image; following binary images correspond to thresholded version with increasing threshold values.*

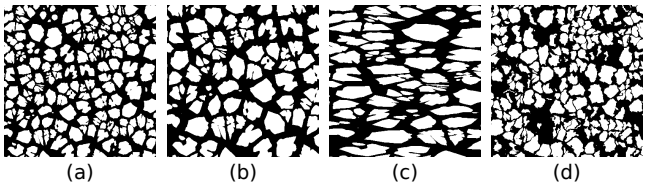

Figure 4: *Spatial transforms are used for normalization: we apply scaling (b), stretching (c), rotation and a $1/f^n$ noise-based spatial warping (d), aiming at simulating natural Brownian motion, a frequent stochastic process in nature. Conversely, distortions due to perspective projection and surface curvature are not considered for normalization.*

4. Source code and algorithm

In the following, we provide a pseudo-code for the PPTBF (see Appendix A). A full GPU implementation is available on our website †. Functions highlighted in red are also implemented in the GPU. These are:

- `brownnoise`: computes a fractal noise with given amplitude falloff factor;

† https://github.com/ASTex-ICube/semiproctex

- `genPointSet`: generates a point process according to a given tesselation, it provides a list (arrays) of closest rectangular cells and points inside these cells. The number of generated neighboring cells is returned.
- `nthclosest`: orders the points according to closest distance. Result is an indirection table, where element 0 is the index of the closest point, element 1 the index of second-closest point, etc.
- `seeding`, `seeding2` and `rand`: implement the pseudo random number generator. `seeding2` is used to allow the generation of a novel series of numbers, uncorrelated from the series initialized with seeding at the same spatial location.
- `beziercell` and `cellborder`: compute intersections with Voronoi -cells that a given location x belongs to, the former applying the Bezier-based cell smoothing described in the paper. Both return a distance value.
- `p-norm`: computes a Minkowski distance.

Appendix A: PPTBF pseudo-code

See next page.

```
// POINT PROCESS TEXTURE BASIS FUNCTION PSEUDO-CODE

// This pseudo code slightly differs from the actual GPU implementation:
// some parameters being tuned on GPU to improve performance
// PPTBF Parameters provided in the supplementals match the GPU implementation
// that will be made publicly available

float compute_pptbf ( vec2 x, // where to compute PPTBF

       // deformation and normalization parameters
       float zoom, float rotation_angle, float rescalex,
       float ampli[3], // Brownian distorsion parameters

       // point set parameters
       int tile_type,
       float jitter,

       // window function parameters
       float normblend, // ω
       float arity, // nv control points for Bezier
       float larp,  // λ window anisotropy
       float wsmooth, // lc window smoothing
       float norm, float sigw1, float sigw2, // ||.|| and σj

       // feature function parameters
       int mixture, // mixture model
       float winfeatcorrel, // correlation with window
       float feataniso, // η anisotropy
       int Jmin, int Jmax,
       float freq, // ϕ
       float thickness, float curvature, float deltaorient // τ,κ,θ
       float normfeat, // ||.||f
       float sigcos, float sigcosvar, // σf and σfvar
       )
{
// storing tessellation cells and point distributions
vec2 c[MAX_NEIGH_CELLS]; // cells lower left corner coords
vec2 d[MAX_NEIGH_CELLS]; // cells size
vec2 p[MAX_NEIGH_CELLS]; // random points
int mink[MAX_NEIGH_CELLS];

       //----------------
       // [1] Brownian Deformation
       //----------------

x = x + amp[0] * brownnoise(amp[1]*x*zoom, amp[2]) ;

       //--------------------
       // [2] Model Transform: scale x, rotation and zoom
       //--------------------

x = x * mat2(cos(rotation_angle),-sin(rotation_angle),
                sin(rotation_angle),cos(rotation_angle))*zoom;

       //------------------
       // [3] Point Process
       //------------------

int npp = genPointSet(x, tile_type, jitter, p, c, d);
// order according to nth closest: result is in table mink[]
nthclosest(mink, npp, x, p, c, d, larp, norm);

       //-------------------------
       // [4] PPTBF = PP x ( W F )
       //-------------------------

float pptbfvv = 0.0f; // final value, to be computed and returned
```

```
float priomax = -1.0f; // initial lowest priority for mixture
float minval = -1000.0f; // initial value for max mixture

for (int k = 0; k < npp; k++) // for each tessellation cell
{
  // init PRNG at cell center
  seeding(p[mink[k]]);
  float bezierangularstep = 2.0 * M_PI / arity;
  float bezierstartangle = bezierangularstep * rand();
  int J = Jmin + (int)((float)(Jmax - Jmin)*rand();

            //-----------------------
            // [5] Window Function: W
            //-----------------------

  // window_1: cellular basis window
  float cval = 0.0;
  if (k == 0) // only inside Voronoi cell
  {
    float smoothdist = beziercell(mink[0],x,c,d,p,
                           bezierangularstep, bezierstartangle);
    float cdist = cellborder(mink[0],x,c,d,p);
    cval = mix(smoothdist,cdist,wsmooth);
  }
  float w1 = normblend * (exp((cv - 1.0)*sigw1) - exp(-1.0*sigw1));
  if (w1<0) w1=0;

  // window_2: overlapping basis window
  float sddno = p-norm(x - p[mink[k]]);
  // empirical constant for clamping gaussian, depending on tile type
  float footprint = 1.5
  if (tile_type >= 10) footprint *= 0.4;
  // compute w2
  float w2 = (1.0 - normblend) * exp(-sigw2 * sddno) - exp(-sigw2* footprint))
  if (w2<0) w2=0;

            //----------------------
            // [7] Feature Function
            //----------------------

  float feat = 0.0; // feature function value to be computed

  // stringed Gaussian parameters
  float mu[MAX_G], dif[MAX_G];
  float theta[MAX_G], prior[MAX_G], sigb[MAX_G];
  float valb[MAX_G]; // for amplitude

  // init PRNG, decorelated from window seed
  seeding2(p[mink[k]]);
  for (int i = 0; i < J; i++)
  {
    prior[i] = rand();
    valb[i] = rand();
    mu[i] = c[mink[k]] + (0.5+0.5*rand()) * d[mink[k]];
    // shift mu according to correlation
    mu[i] = mix(p[mink[k]],mu[i], winfeatcorrel);
    // orient Gabor stripes
    dif[i] = (x - mu[i]) / d[mink[k]];
    theta[i] = deltaorient * rand();
    sigb[i] = sigcos * (1.0 + sigcosvar*rand());
    // apply rotation, anisotropy and curliness
    vec2 dd = mat2(cos(theta[i]),-sin(theta[i]),
                   sin(theta[i]),cos(theta[i])) * dif[i];
    dd.y /= featantiso;
    float xfeat = sqrt(dd.x * dd.x * curvature * curvature + dd.y * dd.y);
    // compute stringed gaussian value
    float ff = 0.5 + 0.5 * cos(π * freq * xfeat);
    ff = pow(ff, 1.0 / (0.0001 + thickness)); // avoids division by zero
```

```
    float fdist = p-norm(dd,normfeat) / (footprint / sigb[i]);
    // apply mixture
    switch (mixture) {
      case 1:
        float amp = valb[i] < 0.0 ? -0.25 + 0.75 * valb[i] : 0.25 + 0.75 * valb[i];
        feat += ff * amp * exp(-fdist);
        break;
      case 2:
      case 3:
        feat += ff * exp(-fdist);
        break;
      case 4:
        if (priomax < prior[i] && fdist < 1.0 && ff>0.5)
         {
           priomax = prior[i];
           pptbfvv = 2.0*(ff - 0.5) * exp(-fdist);
         }
        break;
      case 5:
        float ww = ff* exp(-fdist);
        if (minval < ww) { pptbfvv = ww;  minval = ww; }
        break;
      default: feat = 1.0;
      }
    // normalization according to mixture model
    if (mixture == 1) feat = 0.5 * feat + 0.5;
    if (mixture == 2) feat /= float(J);
    if (mixture == 3) feat = 1.0-feat;

    // add contribution except for max operators
    if (mixture < 4) pptbfvv += (w1 + w2) * feat;
    }
return pptbfvv;
}
```