Appendix A: PPTBF pseudo-code.

```
// 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, // l꜀ 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
```

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

```
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 /= feataniso;
    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
```

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.

```
    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;
}
```

© 2020 The Author(s)
Computer Graphics Forum © 2020 The Eurographics Association and John Wiley & Sons Ltd.