Appendix A: Direct Volume Rendering in Diderot

The base program for the direct volume rendered figures is below. It was used as-is for Fig. 1. The program comments should support understanding its operation; some additional explanation follows.

```
1   input vec3 camEye ("camera look-from point");
2   input vec3 camAt ("camera look-at point");
3   input vec3 camUp ("camera pseudo-up vector");
4   input real camNear ("at-relative near clip distance");
5   input real camFar ("at-relative far clip distance");
6   input real camFOV ("vertical field-of-view angle");
7   input bool camOrtho ("orthographic (not perspective)") = false;
8   input int iresU ("image # horizontal samples");
9   input int iresV ("image # vertical samples");
10  input real rayStep ("ray inter-sample distance");
11  input real refStep ("opacity reference step length");
12  input real transp0 ("early ray stopping transparency") = 0.005;
13  input real thick ("approximate thickness of feature");
14  input real fStrTh ("feature strength threshold");
15  input real fMaskTh ("feature mask threshold") = 0;
16  input real fBias ("Bias in feature strength computing") = 0.0;
17  input real maxAlpha ("maximum opacity of feature");
18  input vec4 phong ("Phong Ka Kd Ks Sp") = [0.1, 0.7, 0.2, 100];
19  input vec3 litdir ("view-space light direction") = [-1,-2,-1];
20  input vec3 mcNear ("color at near clip plane") = [1,1,1];
21  input vec3 mcFar ("color at far clipping plane") = [1,1,1];
22  input real isoval ("which isosurface to render");
23  input image(3)[] vol ("data to render") = image("vol.nrrd");
24  input image(1)[3] cmap ("scalar colormap") = image("cmap.nrrd");
25  input vec2 cmmm ("min,max colormap range") = [0,0];
26
27  field#2(3)[] F = bspln3 ⊛ clamp(vol);
28  field#0(3)[] Fcm = F;  // colormap scalar field itself
29  field#0(1)[3] CM = tent ⊛ clamp(cmap); // 1-D colormap field
30
31  // Isosurface-specificity limited to these four functions
32  function vec3 fStep(vec3 x) =
33     (isoval - F(x))*∇F(x)/(∇F(x)•∇F(x));
34  function real fStrength(vec3 x) = |∇F(x)|;
35  function real fMask(vec3 x) = F(x);
36  function bool fTest(vec3 x) = true;
37
38  // Computing ray parameters and view-space basis
39  vec3 camN = normalize(camAt - camEye);// N: away from eye
40  vec3 camU = normalize(camN × camUp);  // U: right
41  vec3 camV = camN × camU;              // V: down
42  real camDist = |camAt - camEye|;
43  real camVmax = tan(camFOV*π/360)*camDist;
44  real camUmax = camVmax*iresU/iresV;
45  real camNearVsp = camNear + camDist; // near clip, view space
46  real camFarVsp = camFar + camDist;   // far clip, view space
47
48  // Convert light direction from view-space to world-space
49  vec3 litwsp = transpose([camU,camV,camN])•normalize(litdir);
50
51  // Core opacity function is a capped tent function
52  function real atent(real a0, real d)
53    = a0*clamp(0, 1, 1.5*(1 - |d|/thick));
54
55  function bool posTest(vec3 x)
56    = (inside(x, F)              // in field
57       && fStrength(x) > fStrTh // possibly near feature
58       && fMask(x) >= fMaskTh   // meets feature mask
59       && fTest(x));            // passes addtl feature criterion
60
61  // Each strand renders one ray through (rayU,rayV) on view plane
62  strand raycast(int ui, int vi) {
63    // Compute geometry of ray through pixel [ui,vi]
64    real rayU = lerp(-camUmax, camUmax, -0.5, ui, iresU-0.5);
65    real rayV = lerp(-camVmax, camVmax, -0.5, vi, iresV-0.5);
66    real rayN = camNearVsp;
67    vec3 UV = rayU*camU + rayV*camV;
68    vec3 rayOrig = camEye + (UV if camOrtho else [0,0,0]);
69    vec3 rayVec = camN + ([0,0,0] if camOrtho else UV/camDist);
70
71    // Opacity correction is via alphaFix; distance between
72    // ray samples is |rayVec|*rayStep
73    real alphaFix = |rayVec|*rayStep/refStep;
74    vec3 eyeDir = -normalize(rayVec);
75
76    // Unpack Phong parameters
77    real phKa = phong[0]; real phKd = phong[1];
78    real phKs = phong[2]; real phSp = phong[3];
79
80    output vec4 rgba = [0,0,0,0]; // ray output
81    vec3 rgb = [0,0,0]; // ray state is current color ...
82    real transp = 1;    // ... and current tranparency
83
84    update {
85      rayN += rayStep;          // increment ray position
86      if (rayN > camFarVsp) {   // ray passed far clip plane
87        stabilize;
88      }
89      vec3 pos = rayOrig + rayN*rayVec; // ray sample position
90      if (!posTest(pos)) {
91        continue;
92      }
93
94      vec3 step = fStep(pos);          // step towards feature
95      real aa = atent(maxAlpha, |step|); // opacity
96      if (aa == 0) { continue; }     // skip if no opacity
97      aa = 1 - (1 - aa)^alphaFix;    // opacity correction
98      vec3 snorm = -normalize(step); // "surface normal"
99      real dcomp = (snorm•litwsp)^2; // two-sided lighting
100     real scomp = |snorm•normalize(eyeDir+litwsp)|^phSp
101                  if phKs != 0 else 0.0;
102
103     // simple depth-cueing
104     vec3 dcol = lerp(mcNear, mcFar, camNearVsp, rayN, camFarVsp);
105     vec3 mcol = CM(lerp(0, 1, cmmm[0], Fcm(pos+step), cmmm[1]))
106                 if (cmmm[0] != cmmm[1]) else [1,1,1];
107     // light color is [1,1,1]
108     rgb += transp*aa*((phKa + phKd*dcomp)*modulate(dcol,mcol)
109                       + phKs*scomp*[1,1,1]);
110     transp *= 1 - aa;
111     if (transp < transp0) { // early ray termination
112       transp = 0;
113       stabilize;
114     }
115   }
116   stabilize {
117     if (transp < 1) {  // un-pre-multiply opacities
118       real aa = 1-transp;
119       rgba = [rgb[0]/aa, rgb[1]/aa, rgb[2]/aa, aa];
120     }
121   }
122 }
123 initially [ raycast(ui, vi)
124             | vi in 0..iresV-1, ui in 0..iresU-1 ];
```

The renderer is made specific to isosurface with the feature step `fStep` (line 32) and feature strength `fStrength` (line 34) functions. As described in Sec. 3 and demonstrated in Secs. 4 and 5, different feature step and strength functions will repurpose the renderer for different types of features. Vector and tensor field rendering will involve defining some derived scalar field `F` from the multi-variate data, rather than directly creating `F` from the data as in line 27 above. The feature mask function `fMask`, (line 35) described in Sec. 3.3, offers additional tunable control over what parts of a feature are worth seeing, and the test function `fTest` (line 36) is available as a further criterion for feature membership. These are used in the `posTest` function (line 55) function, which used on line 90 to skip over some ray samples.

Appendix B: Particle-based Feature Sampling in Diderot

The base program for the particle-based feature sampling is below. It was used as-is to generate the isosurface sampling seen in Fig. 1f. The program comments should support understanding its operation; additional explanations follow.

```
input real fStrTh ("Feature strength threshold");
input real fMaskTh ("feature mask threshold") = 0;
input real fBias ("Bias in feature strength computing") = 0.0;
input real tipd ("Target inter-particle distance");
/* tipd is the only length or speed variable with data spatial
   units; everything else measures space in units of tipd */
input real mabd ("Min allowed birth distance (> 0.7351)") = 0.75;
input real travMax ("Max allowed travel to or on feature") = 10;
input int nfsMax ("Max allowed # feature steps ") = 20;
// these next three control the Gradient Descent in Energy
input real gdeTest ("Scaling in sufficient decrease test") = 0.5;
input real gdeBack ("How to scale stepsize for backtrack") = 0.5;
input real gdeOppor ("Opportunistic stepsize increase") = 1.2;
input real fsEps ("Conv. thresh. on feature step size");
input real geoEps ("Conv. thresh. on system geometry") = 0.1;
input real mvmtEps ("Conv. thresh. on point movement") = 0.01;
input real rpcEps ("Conv. thresh. on recent pop. changes") = 0.01;
input real pcmvEps ("Motion limit before PC") = 0.3;
input real isoval ("Which isosurface to sample") = 0;
input int verb ("Verbosity level") = 0;
input real sfs ("Scaling (<=1 for stability) on fStep") = 0.5;
input real hist ("How history matters for convergence") = 0.5;
// higher hist: slower change, more stringent convergence test
input int pcp ("periodicity of population control (PC)") = 5;
input vec3{} ipos ("Initial point positions");
input image(3)[] vol ("data to analyze");

field#2(3)[] F = bspln3 ⊛ clamp(vol);

// Isosurface-specificity limited to fDim and these 5 functions
int fDim = 2;
function vec3 fStep(vec3 x) =
  (isoval - F(x))*∇F(x)/(∇F(x)•∇F(x));
function tensor[3,3] fPerp(vec3 x) {
  vec3 norm = normalize(∇F(x));
  return identity[3] - norm⊗norm;
}
function real fStrength(vec3 x) = |∇F(x)|;
function real fMask(vec3 x) = F(x);
function bool fTest(vec3 x) = true;

function bool posTest(vec3 x) =
   (inside(x, F)              // in field
     && fStrength(x) > fStrTh // possibly near feature
     && fMask(x) >= fMaskTh   // meets feature mask
     && fTest(x));            // passes addtl feature criterion

// Each particle wants between nnmin and nnmax neighbors
int nnmin = 6 if (2==fDim) else 2 if (1==fDim) else 0;
int nnmax = 8 if (2==fDim) else 3 if (1==fDim) else 0;

/* Potential function (found with Mathematica) phi(r):
   phi(0)=1, phi(r)=0 for r >= 1, with minima (potential well)
   phi'(2/3)=0 and phi(2/3)=-0.001.  Phi(r) is C^3
   continuous across the well and with 0 for r >= 1. Potential
   well induces good packing with energy minimization. */
function real phi(real r) {
  real s=r-2.0/3;
  return
    1 + r*(-5.646 + r*(11.9835 + r*(-11.3535 + 4.0550625*r)))
  if r < 2.0/3 else
    -0.001 + ((0.09 + (-0.54 + (1.215 - 0.972*s)*s)*s)*s)*s
  if r < 1 else 0;
}
function real phi'(real r) { // phi'(r) = d phi(r) / dr
  real t=3*r-2;
  return
    -5.646 + r*(23.967 + r*(-34.0605 + 16.22025*r))
  if r < 2.0/3 else
    0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))
  if r < 1 else 0;
}
real phiWellRad = 2/3.0;     // radius of potential well
real rad = tipd/phiWellRad; // actual radius of potential support
function real enr(vec3 x) = phi(|x|/rad);
function vec3 frc(vec3 x) = phi'(|x|/rad) * (1/rad) * x/|x|;

// pchist reflects periodicity of PC: pchist^(2*pcp) = hist
real pchist = hist^(1.0/(2*pcp));

int iter = 0;         // iteration counter
real rpc = 1;         // recent population change
int popLast = -1;    // population at last iteration

/* Finds a number in [0,1) roughly proportional to the low 32
   bits of significand of given real x. NOTE: ONLY useful only
   when compiling with --double */
function real urnd(real x) {
  if (x==0) return 0;
  real l2 = log2(|x|);
  real frxp = 2^(l2-floor(l2)-1); // in [0.5,1.0), like frexp(x)
  // use iter to make different values for same x
  return fmod((2^20 + 2*iter)*frxp, 1);
}

// Given vec3 (and iter), a random-ish value uniformly in [0,1)
function real v3rnd(vec3 v)
  = fmod(urnd(v[0]) + urnd(v[1]) + urnd(v[2]), 1);

// Given vec3 (and iter), a big random-ish integer
function real genID(vec3 v) = floor(1000000*v3rnd(v));

/* Is this an iteration in which to do population control (PC)?
   If not, pcIter() returns 0. Otherwise, returns 1 when should
   birth new particles, and -1 when should kill then off. This
   alternation is not due to any language limitations; it just
   plays well with the PC heuristics used here. */
function int pcIter() = ((iter/pcp)%2)*2 - 1
                        if (pcp>0 && iter>0 && 0 == iter % pcp)
                        else 0;

// Strands first find feature, then interact w/ or make neighbors
strand point (vec3 p0, real hh0) {
  output vec3 pos = p0; // current particle position
  real ID = genID(p0);  // strand identifier
  real hh = hh0;        // energy gradient descent stepsize
  vec3 step = [0,0,0];  // energy+feature steps this iter
  bool found = false;   // whether feature has been found
  int nfs = 0;          // number feature steps taken
  real trav = 0;        // total distance traveled
  real mvmt = 1;        // average of recent movement
  real closest = rad;   // distance to closest neighbor
  int born = 0;         // how many particles I have birthed
  bool first = true;    // first time through update
  update {
    if (!posTest(pos)) {
      die;
    }
    if (travMax > 0 && trav > travMax) {  // too much travel
      die;
    }
    if (!found) { // ------------------------ looking for feature
      if (nfsMax > 0 && nfs > nfsMax) {    // too many steps
        die;
      }
      step = sfs*fStep(pos); // one step towards feature
      pos += step;
      mvmt = lerp(|step|/tipd, mvmt, hist);
      if (mvmt > fsEps) {         // still moving
        trav += |step|/tipd;
        nfs += 1;
      } else {          // found feature, prepare for code below
        found = true;
        mvmt = 1;
        trav = 0;
      }
    } else { // ------------------ feature found; minimize energy
      // if feature is isolated points, we're already done
      if (0 == fDim) { stabilize; }
      step = sfs*fStep(pos); pos += step; trav += |step|/tipd;
      real oldE = 0;              // energy at current location
      vec3 force = [0,0,0];       // force on me from neighbors
      int nn = 0;                 // number of neighbors
      foreach (point P in sphere(rad)) {
        vec3 off = P.pos - pos;
        if (|off|/tipd < fsEps && ID <= P.ID) {
          // with 0-D features or unlucky intialization, points
          // can really overlap; point w/ lower ID dies
          die;
        }
        oldE += enr(off);
        force += frc(off);
        nn += 1;
      }
      if (0 == nn) { // else fDim is 1 or 2
        // No neighbors; create one if possible
        if (!( pcIter() > 0 && born < nnmax )) { continue; }
        // Ensure new pos is near feature, for all
        // feature dimensions and orientations
        vec3 noff0 = fPerp(pos)•[tipd,0,0];
        vec3 noff1 = fPerp(pos)•[0,tipd,0];
        vec3 noff2 = fPerp(pos)•[0,0,tipd];
173         vec3 noff = noff0;
174         noff = noff if |noff| > |noff1| else noff1;
175         noff = noff if |noff| > |noff2| else noff2;
176         // noff is now longest of noff0, noff1, noff2
177         vec3 npos = tipd*normalize(noff) + pos;
178         npos += sfs*fStep(npos);
179         if (posTest(pos)) {
180           new point(npos, hh); born += 1;
181         }
182         continue;
183       }
184       // Else I did have neighbors; interact with them
185       vec3 es = hh*fPerp(pos)•force; // energy step along force
186       if (|es| > tipd) {             // limit motion to tipd
187         hh *= tipd/|es|;             // decrease stepsize, step
188         es *= tipd/|es|;
189       }                              // now |es| <= tipd
190       vec3 fs = sfs*fStep(pos+es);   // step towards feature
191       if (|fs|/(fsEps*tipd + |es|) > 0.5) {
192         hh *= 0.5; // feature step too big, try w/ smaller step
193         continue;
194       }
195       vec3 oldpos = pos;
196       pos += fs + es;            // take steps, find new energy
197       real newE = 0;
198       closest = rad;
199       // find mean neighbor offset (mno) to know (opposite)
200       // direction in which to add new particles with PC
201       vec3 mno = [0,0,0];
202       nn = 0;
203       foreach (point P in sphere(rad)) {
204         vec3 off = P.pos - pos;
205         newE += enr(off);
206         closest = min(closest, |off|);
207         mno += off;
208         nn += 1;
209       }
210       mno /= nn;
211       // test the Armijo sufficient decrease condition
212       if (newE - oldE > gdeTest*(pos - oldpos)•(-force)) {
213         // backtrack because energy didn't go down enough
214         hh *= gdeBack;     // try again next time w/ smaller step
215         if (0 == hh) {
216           die; // backtracked all the way to hh=0!
217         }
218         pos = oldpos;
219         continue;
220       }
221       hh *= gdeOppor; // opportunistically increase stepsize
222       step += fs + es;
223       trav += |step|/tipd;
224       mvmt = lerp(|step|/tipd, mvmt, hist);
225       if (|step|/tipd < pcmvEps && pcIter() != 0) {
226         // can do PC only if haven't moved a lot
227         if (pcIter()>0        // this is an iter to add
228             && newE<0         // already in a potential well
229             && nn<nnmin       // have fewer than expected neighbors
230             && born<nnmax) { // haven't birthed too many times
231           vec3 npos = pos - tipd*normalize(mno);
232           npos += sfs*fStep(npos); npos += sfs*fStep(npos);
233           bool birth = true;
234           if (fDim == 2 && nn >= 4) {
235             foreach (point P in sphere(npos, tipd*mabd)) {
236               birth = false; // too close to existing point
237             }
238             if (birth) {
239               // Have nn neighbors: too few (nnmin > nn).
240               // Try adding a new neighbor with a probability
241               // that scales with nnmin-nn.
242               birth = v3rnd(pos) < (nnmin - nn)/real(nnmin);
243             }
244           }
245           if (birth && posTest(npos)) {
246             new point(npos, hh); born += 1;
247           }
248         } else if (pcIter() < 0 && newE > 0 && nn > nnmax) {
249           // Have too many neighbors, so maybe die. If I have
250           // nn neighbors, they probably also have nn neighbors.
251           // To have fewer, that is, nnmax neighbors, we all
252           // die with chance of nn-nnmax out of nn.
253           if (v3rnd(pos) < (nn - nnmax)/real(nn)) {
254             die;
255           }
256         }
257       }
258     } // else found
259     first  = false;
260   } // update
261 }
262 global {
263   int pop = numActive();
264   int pc = 1 if pop != popLast else 0;
265   rpc = lerp(pc, rpc, pchist);
266   bool allfound = all { P.found | P in point.all};
267   real percfound =
268     100* mean { 1.0 if P.found else 0.0 | P in point.all};
269   real meancl = mean { P.closest | P in point.all };
270   real varicl = mean { (P.closest - meancl)^2 | P in point.all };
271   real covcl = sqrt(varicl) / meancl;
272   real maxmvmt = max { P.mvmt | P in point.all };
273   print("======= finished iter ", iter, " w/ ", pop, ")",
274         "; %found=", percfound,
275         "; mean(hh)=", mean { P.hh | P in point.all},
276         "; mean(cl)=", meancl,
277         "; COV(cl)=", covcl,
278         "; max(mvmt)=", maxmvmt,
279         "; pc=", pc,
280         "; rpc=", rpc,
281         "\n");
282   if (allfound            // all particles have found the feature
283       && covcl < geoEps   // and system is geometrically uniform
284       && maxmvmt < mvmtEps  // and nothing's moving much
285       && rpc < rpcEps) { // and pop. hasn't changed recently
286     print("======= Stabilizing ", numActive(), " (iter ", iter, ")",
287           "; COV(cl)=", covcl, " < ", geoEps,
288           "; max(mvmt)=", maxmvmt, " < ", mvmtEps,
289           "; rpc=", rpc, " < ", rpcEps,
290           "\n");
291     stabilize;
292   }
293   iter += 1;
294   popLast = pop;
295 }
296 initially { point(ipos[ii], 1) | ii in 0 .. length(ipos)-1 };
```

As with the direct volume renderer, the code specific to one feature is isolated to one place: the statement of feature dimension `fDim` (line 31), and the feature functions starting on line 32. Relative to the volume renderer, the new feature function is `fPerp` (line 34), which projects onto the orthogonal complement of the possible local feature steps.

Compared with the basic particle system program (Fig. 4), the program is longer and more complex, but the basic structure is the same. There is still a univariate inter-particle potential energy $\phi(r)$, is implemented as `phi` (line 57), which is a piecewise polynomial with a slight potential well at $r = 2/3$. The function is graphed in

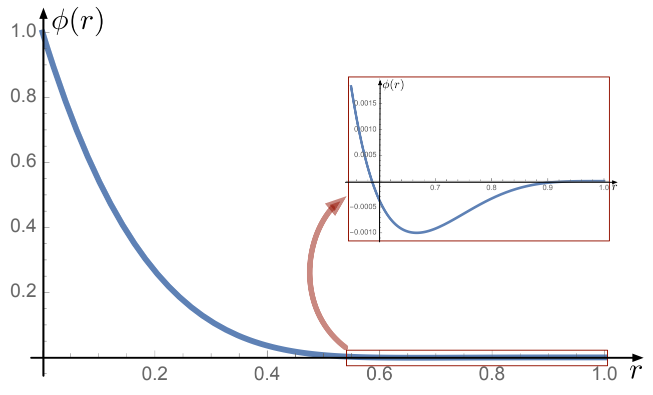

Figure 11: *Graph of inter-particle potential function $\phi(r)$*

Fig. 11, which includes an inset that vertically expands the plot over interval $[0.55, 1]$ to clarify the location and shape of the potential well. The relative shallowness of the potential well compared to height at $r = 0$ ensures that energy minimization separates close particles before it attempts to produce uniform spacing.

The functions over 3D space for energy (`enr` on line 75) and force (`frc` on line 76) are defined as they were in simple Fig. 4

program. The control of the population of the particle system is probabilistic in flavor, using function `v3rnd` (line 97) which generates from a **`vec3`** `v` a value in $[0,1)$ by combining the low-order bits of the X, Y, and Z coordinates of `v` (as exposed by `urnd` on line 88) with the current program iteration count. The current version of Diderot lacks a pseudo-random number generator. The same `v3rnd` is used in the `genID` function (line 101) used to assign to each strand a number (hopefully unique), which proves useful for debugging. A unique per-strand identifier that is thread-safe and stable across iterations is currently not available in Diderot. The periodicity of considering to add or kill particles is controlled by `pcIter` (line 108).

As in the simple particle system (Fig. 4), each program strand computes the position of one particle. Each particle starts (with `found=`**`false`**, line 118) looking for the feature of interest with repeated `fSteps` (lines 132 through 146) while ignoring other particles, after which (lines 147 through 258) particles interact with each other to produce a uniform feature sampling. This second phase includes careful mechanisms for population control. If particles have no neighbors (lines 165 through 183), an effort is made to create a new neighbor close to the feature, using `fPerp`. Computing energy at the updated location (lines 203 through 210) includes computing a mean offset to neighbors `mno`, which is used later (line 231) as part of determining where to try add a new particle in case of under-population. Because the $\phi(r)$ function in the minimal Fig. 4 particle system program was purely repulsive, the last energy gradient descent direction could play that role (Fig. 4 line 73), but here the $\phi(r)$ includes a potential well, so the geometric information in `mno` is useful. If the particle has not predictably moved downhill in energy (line 212), it backtracks and tries again on the next iteration.

Otherwise (line 221), with predictable energy descent, the records of recent motion are updated (line 222), and, if recent motion is small, population control is considered (local estimates of particle density mean less if the system is rapidly moving). Precautions are taken to ensure that the intended location of the a new particle are not too close to an existing one, via the minimum allowed birth distance `mabd` parameter (input line 7, used line 235). This parameter is subtle: if too high, significant holes are never filled in, and if too low, then the pentagonal arrangements of points that may appropriately minimize energy on higher curvature surfaces may trigger the birth of multiple particles, each trying to create a local hexagonal packing (wherein every particle will see $nn_{\text{min}} = 6$ neighbors). Fig. 12 illustrates the geometric reasoning involved in setting `mabd`. If particles, separated by S, have formed a pentagon, then if one adds a new particle at distance S, it will have distance D from another particle on the other side of the pentagon; $D/S \approx 0.735085$. Setting `mabd` higher than this (0.75 works in our experience) prevents pentagonal holes from triggering excessive births. Subsequent meshing can fill the whole by adding two edges and three triangles.

The chances of creating a new particle (if the `mabd` test passes, line 242) or of a particle exiting the computation (line 253) depend on the relationship between the number of neighbors `nn` and the target range of neighbor numbers [`nnmin`,`nnmax`]. The intent is that after one or two periods of population control, the system has roughly the correct number of particles and can proceed to distribute them in a uniform way. While this code with these parameter settings worked adequately to produce our current results, we hope that further computational and geometric analysis can demonstrate the theoretical stability and robustness of the method.

In the final part of the program, the global update (line 262), the particle system state is measured to test for convergence (line 282), which includes tests on the recent stability of particle position and number, as well as their spatial uniformity, as measured by the coefficient-of-variation of distances to interacting neighbors.

$$\frac{D}{S} = \sqrt{\frac{H^2}{S^2} - \frac{2H}{S} + \frac{5}{4}} = \frac{1}{\sqrt{1 + \sqrt{\frac{1}{10}\left(5+\sqrt{5}\right)}}} \approx 0.735085$$

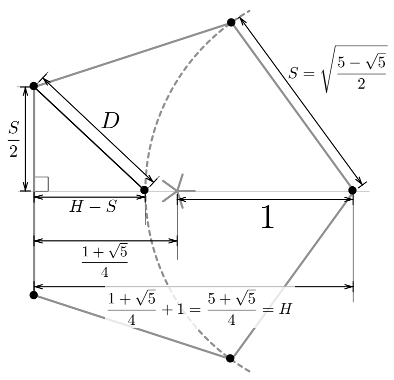
$S = \sqrt{\frac{5-\sqrt{5}}{2}}$
D
$\frac{S}{2}$
$H - S$
1
$\frac{1+\sqrt{5}}{4}$
$\frac{1+\sqrt{5}}{4} + 1 = \frac{5+\sqrt{5}}{4} = H$

Figure 12: *Geometric derivation of lower bound on* `mabd` *parameter to avoid filling pentagonal holes in sampling*

Appendix C: Human-readable Diderot intermediate representation

The ability of the Diderot compiler to generate code that computes higher-order derivatives of vector and tensor fields has enabled our work to date. How any compiler converts the surface programming langauge into working code requires multiple stages of internal or intermediate representation. We thought it might be interesting to see what the Diderot compiler is doing with the expressions associated with extremal features, by modifying the (open-source) compiler to print some of its intermediate representations. We show here human-readable expressions for gradient and Hessian of the Parallel Vector operator used for many vector field features [PR99].

If we consider two 3D vector fields $\mathbf{a}(\mathbf{x})$ and $\mathbf{b}(\mathbf{x})$ (these two letters are more easily distinguished than the standard $\mathbf{u}(\mathbf{x})$ and $\mathbf{v}(\mathbf{x})$), the Parallel Vector Operator (PVO) $\mathbf{a}||\mathbf{b}$ is true at points $\mathbf{x}$ where $\mathbf{a}(\mathbf{x})$ is parallel to $\mathbf{b}(\mathbf{x})$, i.e.

$$(\mathbf{a}||\mathbf{b})(\mathbf{x}) \;\Leftrightarrow\; P(\mathbf{x}) = \frac{\mathbf{a}(\mathbf{x})}{|\mathbf{a}(\mathbf{x})|} \cdot \frac{\mathbf{b}(\mathbf{x})}{|\mathbf{b}(\mathbf{x})|} = \pm 1 \tag{24}$$

Our approach to visualizing or extracting $\mathbf{a}||\mathbf{b}$ involves finding the Newton step towards $\mathbf{a}||\mathbf{b}$. Since $\mathbf{a}||\mathbf{b}$ are particular ridge and valley lines of $\mathbf{a}\cdot\mathbf{b}/(|\mathbf{a}||\mathbf{b}|)$ (where the height is $+1$ and -1, respectively), we need the gradient and Hessian of $(\mathbf{a}/|\mathbf{a}|)\cdot(\mathbf{b}/|\mathbf{b}|)$ to compute the Newton step with (12) of Sec. 3.1.

We modified the Diderot compiler to learn expressions for these derivatives, by printing LaTeXor Unicode formattings of the intermediate representation. Starting with a minimal program to evaluate once the gradient of the PVO:

```
1  input image(3)[3] A;
2  input image(3)[3] B;
3  field#2(3)[3] a = bspln3 ⊛ A;
4  field#2(3)[3] b = bspln3 ⊛ B;
5
6  field#2(3)[] P = (a/|a|)•(b/|b|); // the PVO
7
8  strand f(int i) {
9    output tensor[3] r = ∇P([0,0,0]);
10   update {
11     stabilize;
12   }
13 }
14 initially [ f(i) | i in 0..0];
```

Our modified compiler generated:

$$\frac{((A \bullet \nabla \otimes B) + (B \bullet \nabla \otimes A))}{(|A| * |B|)} \tag{25}$$

$$-\Big(\frac{((B \bullet A) * (A \bullet \nabla \otimes A))}{(|A| * |B| * (A \bullet A))} + \frac{((B \bullet A) * (B \bullet \nabla \otimes B))}{(|B| * |A| * (B \bullet B))}\Big), \tag{26}$$

which we manually post-processed to find:

$$\nabla P = \frac{\mathbf{a}\cdot\nabla\otimes\mathbf{b} \;+\; \mathbf{b}\cdot\nabla\otimes\mathbf{a} \;-\; \mathbf{a}\cdot\mathbf{b}\left(\frac{\mathbf{a}\cdot\nabla\otimes\mathbf{a}}{\mathbf{a}\cdot\mathbf{a}} + \frac{\mathbf{b}\cdot\nabla\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\right)}{|\mathbf{a}||\mathbf{b}|}. \tag{27}$$

We were not previously familiar with this expression of ∇P, which (to first order) points towards (or away from) where $\mathbf{a}$ and $\mathbf{b}$ are parallel. Terms like $\mathbf{a}\cdot\nabla\otimes\mathbf{b}$ are the Jacobian $\nabla\otimes\mathbf{b}$ of $\mathbf{b}$, contracted on the left by $\mathbf{a}$, which can be thought of as a sum over the rows of $\nabla\otimes\mathbf{b}$, weighted by the components of $\mathbf{a}$. The ∇P expression could also be derived by hand, but it was a nearly automatic side-effect of our modified Diderot compiler. The expression for ∇P is symmetric in switching $\mathbf{a}$ and $\mathbf{b}$, which is reassuring.

For comparison, Van Gelder and Pang, also interested in iterative methods to extract PVO features, derive (with a page of careful explanation) this condition for a step $\boldsymbol{\varepsilon}$ from $\mathbf{x}$ such that $\mathbf{x}+\boldsymbol{\varepsilon}$ satisfies $\mathbf{a}||\mathbf{b}$ (c.f. (29) in [GP09]):

$$\mathbf{q} + \left(\mathbf{I} - \frac{\mathbf{b}\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{v}}\right)(\nabla\otimes\mathbf{a} - s\nabla\otimes\mathbf{b})\boldsymbol{\varepsilon}$$

$$-\left(\frac{\mathbf{b}\otimes\mathbf{q}}{\mathbf{b}\cdot\mathbf{b}}\nabla\otimes\mathbf{b} + \frac{\mathbf{q}\otimes\mathbf{a}}{\mathbf{a}\cdot\mathbf{a}}\nabla\otimes\mathbf{a}\right)\boldsymbol{\varepsilon} = \mathbf{0} \tag{28}$$

where

$$\mathbf{q} = \left(\mathbf{I} - \frac{\mathbf{b}\otimes\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\right)\mathbf{a} \tag{29}$$

is the component of $\mathbf{a}$ orthogonal to $\mathbf{b}$. The authors then describe how $\boldsymbol{\varepsilon}$ may then by computed as the solution to a system of equations as part of an iterative search. They chose a mathematical formulation that is not symmetric in switching $\mathbf{a}$ and $\mathbf{b}$.

We were curious if our modified Diderot compiler could produce a human-readable expression for the Hessian of $P(\mathbf{x}) = \frac{\mathbf{a}(\mathbf{x})}{|\mathbf{a}(\mathbf{x})|}\cdot\frac{\mathbf{b}(\mathbf{x})}{|\mathbf{b}(\mathbf{x})|}$, which is inverted as to compute, via (12), the feature step of our approach. By changing line 9 in the program above to include `r = ∇⊗∇P([0,0,0]);` our modified compiler generated a lengthy expression:

$$\begin{aligned}
&\frac{(((\nabla\otimes A)^T \bullet \nabla\otimes B) + (A\bullet\nabla\otimes\nabla\otimes B) + ((\nabla\otimes B)^T \bullet \nabla\otimes A) + (B\bullet\nabla\otimes\nabla\otimes A))}{(|B| * |A|)} \\
&+ \frac{(((B\bullet A) * ((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A))) + (2 * (B\bullet A) * ((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A))))}{(|B| * |A| * ((A\bullet A))^2)} \\
&+ \frac{(((B\bullet A) * ((B\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes A))) + ((B\bullet A) * ((A\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes B))))}{((B\bullet B) * |A| * |B| * (A\bullet A))} \\
&+ \frac{((B\bullet A) * ((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B)))}{(|B| * |A| * ((B\bullet B))^2)} + \frac{(2 * |B| * (B\bullet A) * ((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B)))}{(|A| * (B\bullet B) * ((B\bullet B))^2)} \\
&-\Big(\frac{((((A\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes A))) + (((A\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes B))) + ((B\bullet A) * ((\nabla\otimes A)^T \bullet \nabla\otimes A)) + ((B\bullet A) * (A\bullet\nabla\otimes\nabla\otimes A)) + (((A\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes A))) + (((B\bullet\nabla\otimes A)\otimes(A\bullet\nabla\otimes A))))}{(|B| * |A| * (A\bullet A))} \\
&+ \frac{((((B\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes A))) + (((A\bullet\nabla\otimes B)\otimes(B\bullet\nabla\otimes B))) + (((B\bullet\nabla\otimes B)\otimes(A\bullet\nabla\otimes B))) + (((B\bullet\nabla\otimes A)\otimes(B\bullet\nabla\otimes B))))}{(|A| * |B| * (B\bullet B))} \\
&+ \frac{((|B| * (B\bullet A) * ((\nabla\otimes B)^T \bullet \nabla\otimes B)) + (|B| * (B\bullet A) * (B\bullet\nabla\otimes\nabla\otimes B)))}{(|A| * (B\bullet B) * (B\bullet B))}\Big)
\end{aligned}$$

With some manual post-processing (factoring common terms and regrouping), we develop an expression for the Hessian of P:

$$\nabla\otimes\nabla P = \frac{\begin{array}{c} (\nabla\otimes\mathbf{b})^T\cdot\nabla\otimes\mathbf{a} + (\nabla\otimes\mathbf{a})^T\cdot\nabla\otimes\mathbf{b} + \mathbf{a}\cdot\nabla\otimes\nabla\otimes\mathbf{b} + \mathbf{b}\cdot\nabla\otimes\nabla\otimes\mathbf{a} \\ +\mathbf{a}\cdot\mathbf{b}\left(\frac{3(\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b})}{(\mathbf{b}\cdot\mathbf{b})^2} + \frac{3(\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a})}{(\mathbf{a}\cdot\mathbf{a})^2} + \frac{(\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b})+(\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a})}{(\mathbf{a}\cdot\mathbf{a})(\mathbf{b}\cdot\mathbf{b})}\right) \\ -\frac{\mathbf{a}\cdot\mathbf{b}\big(\mathbf{b}\cdot\nabla\otimes\nabla\otimes\mathbf{b}+(\nabla\otimes\mathbf{b})^T\cdot\nabla\otimes\mathbf{b}\big) + (\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{b}\cdot\nabla\otimes\mathbf{b}) + (\mathbf{b}\cdot\nabla\otimes\mathbf{b})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})}{\mathbf{b}\cdot\mathbf{b}} \\ -\frac{\mathbf{a}\cdot\mathbf{b}\big(\mathbf{a}\cdot\nabla\otimes\nabla\otimes\mathbf{a}+(\nabla\otimes\mathbf{a})^T\cdot\nabla\otimes\mathbf{a}\big) + (\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{a}) + (\mathbf{a}\cdot\nabla\otimes\mathbf{a})\otimes(\mathbf{a}\cdot\nabla\otimes\mathbf{b}+\mathbf{b}\cdot\nabla\otimes\mathbf{a})}{\mathbf{a}\cdot\mathbf{a}} \end{array}}{|\mathbf{a}||\mathbf{b}|} \tag{30}$$

Review of this expression reveals that it too is symmetric in switching $\mathbf{a}$ and $\mathbf{b}$. $\nabla\otimes\nabla\otimes\mathbf{a}$ is the Hessian of vector field $\mathbf{a}$, a third-order tensor that, when right multiplied by offset $\boldsymbol{\varepsilon}$, gives the local change in the Jacobian. While it would also be possible to derive $\nabla\otimes\nabla P$ by hand, the automated operation of a compiler may be more trustworthy. We show this expression for $\nabla\otimes\nabla P$ to demonstrate functionality that is otherwise hidden inside the Diderot compiler, and to document a complicated formula that others may find useful if implementing Newton steps towards PVO features without Diderot.

Appendix D: Utility programs in Diderot

We we include, for the sake of completeness, other Diderot programs and functions that were used to compute results or generate figures.

D.1. Finding 1D column-space (`col1span`)

```
// finds vector spanning 1D columnspace
function vec3 col1span(tensor[3,3] m) {
  vec3 ret = [0,0,0];
  vec3 c0 = m[:,0]; // extract columns
  vec3 c1 = m[:,1];
  vec3 c2 = m[:,2];
  vec3 c = c0;
  // learn which column is longest
  int which = 0;
  if (|c1| > |c|) { c = c1; which = 1; }
  if (|c2| > |c|) { c = c2; which = 2; }
  // starting with longest column, add in other columns,
  // negating as needed to get longest (most accurate) sum
  if (0 == which) {
    ret = c0;
    ret += c1 if c1•c0 > 0 else -c1;
    ret += c2 if c2•c0 > 0 else -c2;
  } else if (1 == which) {
    ret = c1;
    ret += c0 if c0•c1 > 0 else -c0;
    ret += c2 if c2•c1 > 0 else -c2;
  } else { // 2 == which
    ret = c2;
    ret += c0 if c0•c2 > 0 else -c0;
    ret += c1 if c1•c2 > 0 else -c1;
  }
  // normalize result if possible
  return normalize(ret) if |ret|>0 else [0,0,0];
}
```

The above function is used as part of surface crease line rendering (Sec. 5.3), to find the single eigenvector of a symmetric 3×3 matrix associated with the sole non-zero eigenvalue. This amounts to finding a vector that spans the column space of the matrix, which the above function does by finding the longest possible sum of (possibly negated) columns in the given matrix, and then normalizing.

D.2. Finding edges between particles (`edge.diderot`)

```
1  input vec3{} ipos ("vertex positions") = load("pos.nrrd");
2  input real rad ("radius within which verts are edge-connected");
3
4  strand point (int ii, vec3 pp) {
5    // the output of this program is what it print()s,
6    // rather than this "output" variable foo.
7    output real foo=0;
8    int ID = ii;    // record our index in vert list
9    vec3 pos = pp; // record spatial position
10   update {
11     // the sphere() test implicitly depends on pos
12     foreach (point P in sphere(rad)) {
13       if (ID < P.ID) {
14         // only report each edge once
15         print(ID, "\n", P.ID, "\n");
16       }
17     }
18     stabilize;
19   }
20 }
21
22 initially { point(ii, ipos[ii]) | ii in 0 .. length(ipos)-1 };
```

The above utility program is used for the first stage of meshing feature sampling results systems (Sec. 4.3): connecting neighboring vertices together. Because the particle system tends to produce very uniform samplings at and near convergence, the test for whether two vertices (as represented by two particles) should be considered edge-connected is reduced to knowing if they interacted in the last iteration. Because for this work we have not yet attempted to vary sampling density based on feature characteristics, this is in turn equivalent to asking whether two particles are within the potential function $\phi(r)$ support of each other. Assuming the $\phi(r)$ described in Appendix B, with its potential well at $r = 2/3$, the radius `rad` given to on line 2 should be $3/2$ of the target inter-particle distance (`tipd`, Appendix B line 4) used for particle system computation. A k-d tree created by Diderot run-time based on the special `pos` position variable (line 9) ensures that the **`sphere`** test (line 12) is executed efficiently.

D.3. PostScript mesh drawing (`epsdraw.diderot`)

The program below is included for the sake of completeness since it is used for figure generation (Fig 1 bottom row, Fig. 3(b,c,d), and Fig. 5). It produces a PostScript depiction of small particle systems and their meshes, by computing world-to-view and view-to-screen transformations via homogeneous coordinates. With its ability to label all edges, vertices, and faces in a vector graphics output, it was used for debugging the Appendix B particle system program, and its subsequent meshing. This is not, however, a typical or especially informative Diderot program. Like `edge.diderot` above, the useful output of this program is via its many `print` statements, rather than typical per-strand computed output. Diderot currently has no means of sorting strands based on computed results, so the PostScript commands to draw each element are printed to a single line of text, which starts with "Z `pop`" where Z is screen depth. Sorting these lines as a post-process ensures that PostScript will draw closer elements after (on top of) further elements.

```
input vec3{} ipos ("point positions");
input int{} edg ("edges as pairs of point indices");
input int{} tri ("triangles as triplets of point indices");
int pntNum = length(ipos);
int edgNum = 0 if (edg[0] == 0 && edg[1] == 0)
             else length(edg)/2;
int triNum = 0 if (tri[0] == 0 && tri[1] == 0 && tri[2] == 0)
             else length(tri)/3;
input image(3)[] img ("data to analyze") = image("vol.nrrd");
input vec3 camEye ("camera look-from point") = [6, 9, 2];
input vec3 camAt ("camera look-at point") = [0, 0, 0];
input vec2 clasuv ("Camera Look-at Shift at along U,V") = [0,0];
input vec3 camUp ("camera pseudo-up vector") = [0, 0, 1];
input real camNear ("at-relative near clip distance") = -3;
input real camFar ("at-relative far clip distance") = 3;
input real camFOV ("vertical field-of-view angle") = 15;
input bool camOrtho ("orthographic (not perspective)") = false;
input int iresU ("image # horizontal samples") = 640;
input int iresV ("image # vertical samples") = 480;
input real clwid ("circle line width (in world space!)") = 0.01;
input real elwid ("edge line width (in screen space!)") = 0.1;
input real revth ("draw reversed edges this much thicker") = 6;
input bool cfill ("should fill circle") = true;
input bool bvcull ("back vertex culling") = false;
input real label ("if > 0, font size for labeling things") = 0;
input real crd ("circle radius");
input real drd ("dot radius");
input real frgray ("front-facing gray") = 0.3;
input real egray ("edge gray") = 0;
input real bkgray ("back-facing gray") = 0.8;
input real trigray ("triangle gray") = 0.8;
input real scl ("scaling") = 120;
/* this string identifies what kind of feature should be drawn,
   which matters for choosing how to determine the apparent
   orientation of the disc used to indicate each vertex */
input string feat ("FEAT-ISO, FEAT-RSF, FEAT-VSF, or FEAT-RLN");

// computing ray parameters and view-space basis
vec3 camN_ = normalize(camAt - camEye);// N: away from eye
vec3 camU_ = normalize(camN_ × normalize(camUp)); // U: right
vec3 camV_ = camN_ × camU_;                 // V: down
// now with camAtShift
vec3 camN = normalize(camAt + clasuv[0]*camU_
                      + clasuv[1]*camV_  - camEye);
vec3 camU = normalize(camN × normalize(camUp));
vec3 camV = camN × camU;
real camDist = |camAt + clasuv[0]*camU_
                + clasuv[1]*camV_  - camEye|;
real camVmax = tan(camFOV*π/360)*camDist;
real camUmax = camVmax*iresU/iresV;
real camNearV = camNear + camDist;   // near clip, view space
real camFarV = camFar + camDist;     // far clip, view space

real hght = 2*camVmax;
real width = hght*iresU/iresV;

// determine view transforms
tensor[4,4] WtoV = [
 [camU[0], camU[1], camU[2], -camU●camEye],
 [camV[0], camV[1], camV[2], -camV●camEye],
 [camN[0], camN[1], camN[2], -camN●camEye],
 [0, 0, 0, 1]];
tensor[4,4] perspVtoC = [
 [2*camDist/width, 0, 0, 0],
 [0, 2*camDist/hght, 0, 0],
 [0, 0, (camFarV+camNearV)/(camFarV-camNearV),
  -2*camFarV*camNearV/(camFarV-camNearV)],
 [0, 0, 1, 0]];
tensor[4,4] orthoVtoC = [
 [2/width, 0, 0, 0],
 [0, 2/hght, 0, 0],
 [0, 0, 2/(camFarV-camNearV),
  -(camFarV+camNearV)/(camFarV-camNearV)],
 [0, 0, 0, 1]];
tensor[4,4] VtoC = orthoVtoC if camOrtho else perspVtoC;
tensor[4,4] CtoS = [
 [scl*camUmax, 0, 0, 0],
 [0, scl*camVmax, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]];

field#2(3)[] F = bspln3 ⊛ clamp(img);

// undo homogeneous coords
function vec3 unh(vec4 ch) =
  [ch[0]/ch[3], ch[1]/ch[3], ch[2]/ch[3]];
// convert to homogeneous coords
function vec4 hom(vec3 c) = [c[0], c[1], c[2], 1];
// how to approximate surface "normal"
function vec3 snorm(vec3 p) {
  vec3 ret=[0,0,0];
  if (feat == "FEAT-ISO") {
      ret = normalize(-∇F(p));
  } else if (feat == "FEAT-RSF") {
      ret = evecs(∇⊗∇F(p)){2};
  } else if (feat == "FEAT-VSF" || feat == "FEAT-RLN") {
      ret = evecs(∇⊗∇F(p)){0};
  } else if (feat == "FEAT-CTP") {
      ret = -camN;
  } else {
      ret = [nan,nan,nan];
  }
  return ret;
}
bool snsgn = true  if (feat == "FEAT-ISO") else
             false if (feat == "FEAT-RSF") else
             false if (feat == "FEAT-VSF") else
             false if (feat == "FEAT-RLN") else
             false;

strand draw (int ii) {
  output real foo=0;
  update {
    // only one strand prints preamble
    if (ii==0) {
      print("%!PS-Adobe-3.0 EPSF-3.0\n");
      print("%%Creator: Diderot\n");
      print("%%Title: awesome figure\n");
      print("%%Pages: 1\n");
      print("%%BoundingBox: ", -scl*camUmax, " ", -scl*camVmax,
            " ", scl*camUmax, " ", scl*camVmax, "\n");
      print("%%EndComments\n");
      print("%%BeginProlog\n");
      print("%%EndProlog\n");
      print("%%Page: 1 1\n");
      print("gsave\n");
      print(-scl*camUmax, " ", -scl*camVmax, " moveto\n");
      print(scl*camUmax, " ", -scl*camVmax, " lineto\n");
      print(scl*camUmax, " ", scl*camVmax, " lineto\n");
      print(-scl*camUmax, " ", scl*camVmax, " lineto\n");
      print("closepath clip\n");
      print("gsave newpath\n");
      print("1 -1 scale\n");
      if (label > 0) {
        print("/Times-Roman findfont\n");
        print(label, " scalefont setfont\n");
      }
    }
    if (ii <= pntNum-1) {
      /*
      p_: position of center of glyph to draw
      q_: from p, in direction towards eye, but tangent
          (normal to normal); should get the most fore-shortening
      r_: from p, in direction perpendicular to q's offset from p
      _w: world-space coords
      _s: screen-space coords
      */
      vec3 pw = ipos[ii];
      vec3 nw = snorm(pw);
      if (|nw| >= 0) {
        // nn == Nothing along Normal
        tensor[3,3] nn =  identity[3] - nw⊗nw;
        vec3 toeye = normalize(camEye - pw);
        vec3 qo = drd*normalize(nn●toeye);
        vec3 qw = pw + qo;
        vec3 ro = drd*normalize(nw×qo);
        vec3 rw = pw + ro;
        vec3 ps = unh(CtoS●VtoC●WtoV●hom(pw));
        vec3 qs = unh(CtoS●VtoC●WtoV●hom(qw));
        vec3 rs = unh(CtoS●VtoC●WtoV●hom(rw));
        if (-1 <= ps[2] && ps[2] <= 1
            && (!snsgn || !bvcull || nw●toeye > 0)) {
          print(ps[2], " pop ");
          print("gsave ");
          print(ps[0], " ", ps[1], " translate ");
          real gray = frgray if (!snsgn) else
                      frgray if (nw●toeye > 0) else bkgray;
          vec3 rso = [[1,0,0],[0,1,0],[0,0,0]]●(rs - ps);
          vec3 qso = [[1,0,0],[0,1,0],[0,0,0]]●(qs - ps);
          print(180*atan2(rso[1],rso[0])/π, " rotate ");
          print(|rso|, " ", |qso|, " scale ");
          print(gray, " setgray ");
          if (clwid > 0) {
            print(clwid/drd, " setlinewidth ");
            print("0 0 ", crd/drd," 0 360 arc closepath ");
            if (cfill) { print("gsave 1 setgray fill grestore ");}
            print("stroke ");
            if (frgray == gray) {
              print("0 0 1 0 360 arc closepath fill ");
            }
          } else {
            print("0 0 1 0 360 arc closepath fill ");
          }
          print("grestore ");
          print("% vi=", ii, "\n");
          if (label > 0) {
            print(ps[2]-0.1, " pop gsave 0.5 setgray newpath ",
                  ps[0], " ", ps[1],
                  " moveto 1 -1 scale (v", ii, ") show grestore\n");
          }
        }
      }
    } else if (ii <= pntNum+edgNum-1) {
      int ei=ii-pntNum; // edge index
      int pi0 = edg[0 + 2*ei];
      int pi1 = edg[1 + 2*ei];
      if (pi0 != pi1) {
        vec3 pw0 = ipos[pi0];
        vec3 pw1 = ipos[pi1];
        vec3 nw0 = snorm(pw0);
        vec3 nw1 = snorm(pw1);
        if (|nw0| >= 0 && |nw1| >= 0) {
          vec3 toeye0 = normalize(camEye - pw0);
          vec3 toeye1 = normalize(camEye - pw1);
          if (!snsgn || (toeye0●nw0 > 0 && toeye1●nw1 > 0)) {
            vec3 ps0 = unh(CtoS●VtoC●WtoV●hom(pw0));
            vec3 ps1 = unh(CtoS●VtoC●WtoV●hom(pw1));
            real ez = min(ps0[2], ps1[2]);
            if (-1 <= ez && ez <= 1) {
              print(ez, " pop ");
              print(egray, " setgray ",
                    elwid*(1 if pi0 < pi1 else revth),
                    " setlinewidth ",
                    ps0[0], " ", ps0[1], " moveto ", ps1[0],
                    " ", ps1[1], " lineto stroke % ei=",
                    ei, "\n");
              if (label > 0) {
                print(ez-0.1, " pop ");
                vec3 ms = lerp(ps0, ps1, 0.5);
                print("gsave 0.5 setgray newpath ", ms[0],
                      " ", ms[1], " moveto (e", ei,
                      ") 1 -1 scale show grestore\n");
              }
            }
          }
        }
      }
    } else {
      int ti = ii-pntNum-edgNum; // tri index
      int pi0 = tri[0 + 3*ti];
      int pi1 = tri[1 + 3*ti];
      int pi2 = tri[2 + 3*ti];
      if (!(pi0 == pi1 && pi1 == pi2)) { // not a fake triangle
```

```
        vec3 pw0 = ipos[pi0];
        vec3 pw1 = ipos[pi1];
        vec3 pw2 = ipos[pi2];
        vec3 pwm = (pw0 + pw1 + pw2)/3;
        vec3 nwm = snorm(pwm);
        vec3 toeye = normalize(camEye - pwm);
        if (!snsgn || toeye●nwm > 0) {
          vec3 ps0 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw0,0.5)));
          vec3 ps1 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw1,0.5)));
          vec3 ps2 = unh(CtoS●VtoC●WtoV●hom(lerp(pwm,pw2,0.5)));
          real ez = min(min(ps0[2], ps1[2]), ps2[2]);
          if (-1 <= ez && ez <= 1 && trigray <= 1) {
            print(ez, " pop ");
            print(trigray, " setgray ",
                  ps0[0], " ", ps0[1], " moveto ",
                  ps1[0], " ", ps1[1], " lineto ",
                  ps2[0], " ", ps2[1],
                  " lineto closepath fill % ti=", ti, "\n");
          }
        }
      }
    }
    if (ii == (pntNum+edgNum+triNum)-1) {
        print("-2 pop ");
        print("grestore grestore\n");
    }
    stabilize;
  }
}

initially { draw(ii) | ii in 0 .. (pntNum+edgNum+triNum)-1 };
```