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 camN ("camera up vector");
4 input real camNFar ("at-relative near clip distance");
5 input real camNFar ("at-relative far clip distance");
6 input real camFOV ("vertical field-of-view angle");
7 bool camOrtho ("orthographic (not perspective)") = false;
8 input int ileave ("image # horizontal samples");
9 input int ireaV ("image # vertical samples");
10 input real camStep ("ray-step size");
11 input real fMaskTh ("feature mask threshold") = 0;
12 input real fStrTh ("feature strength threshold");
13 input real thick ("approximate thickness of feature") = 0.5;
14 input real maxAlpha ("maximum opacity of feature") = 0;
15 input real alphaFix ("transparency beyond alpha") = 0;
16 input real refStep ("opacity reference step length") = 0.005;
17 input vec3 litdir ("view-space light direction") = [-1,-2,-1];
18 input vec3 phong ("Phong Ka Kd Ks Sp") = [0.1, 0.7, 0.2, 100];
19 input vec3 litwsp ("view-space light direction") = [1,1,1];
20 input vec3 camNearVsp ("color at near clipping plane") = [1,1,1];
21 input vec3 camFarVsp ("color at far clipping plane") = [1,1,1];
22 input real camUmax ("U: right") = 1;
23 input real camVmax ("V: down") = 1;
24 input real camDist ("ray sample position") = 0.0;
25 input bool camOrtho ("orthographic (not perspective)") = false;
26 input real camFar ("at-relative far clip distance") = 1000;
27 input real camNear ("at-relative near clip distance") = 1;
28 input vec3 camEye ("camera look-from point");
29 input vec3 camAt ("camera look-at point");
30 input vec3 camN ("camera up vector");
31 input real camNFar ("at-relative near clip distance");
32 input real camNFar ("at-relative far clip distance");
33 input real camFOV ("vertical field-of-view angle");
34 bool camOrtho ("orthographic (not perspective)") = false;
35 input int ileave ("image # horizontal samples");
36 input int ireaV ("image # vertical samples");
37 input real fMaskTh ("feature mask threshold") = 0;
38 input real fStrTh ("feature strength threshold");
39 input real thick ("approximate thickness of feature") = 0.5;
40 input real maxAlpha ("maximum opacity of feature") = 0;
41 input real alphaFix ("transparency beyond alpha") = 0;
42 input real refStep ("opacity reference step length") = 0.005;
43 input vec3 litdir ("view-space light direction") = [-1,-2,-1];
44 input vec3 phong ("Phong Ka Kd Ks Sp") = [0.1, 0.7, 0.2, 100];
45 input vec3 litwsp ("view-space light direction") = [1,1,1];
46 input vec3 camNearVsp ("color at near clipping plane") = [1,1,1];
47 input vec3 camFarVsp ("color at far clipping plane") = [1,1,1];
48 input real camUmax ("U: right") = 1;
49 input real camVmax ("V: down") = 1;
50 input real camDist ("ray sample position") = 0.0;
51 input bool camOrtho ("orthographic (not perspective)") = false;
52 input real camFar ("at-relative far clip distance") = 1000;
53 input real camNear ("at-relative near clip distance") = 1;
54 input vec3 camEye ("camera look-from point");
55 input vec3 camAt ("camera look-at point");
56 input vec3 camN ("camera up vector");
57 input real camNFar ("at-relative near clip distance");
58 input real camNFar ("at-relative far clip distance");
59 input real camFOV ("vertical field-of-view angle");
60 bool camOrtho ("orthographic (not perspective)") = false;
61 input int ileave ("image # horizontal samples");
62 input int ireaV ("image # vertical samples");
63 input real fMaskTh ("feature mask threshold") = 0;
64 input real fStrTh ("feature strength threshold");
65 input real thick ("approximate thickness of feature") = 0.5;
66 input real maxAlpha ("maximum opacity of feature") = 0;
67 input real alphaFix ("transparency beyond alpha") = 0;
68 input real refStep ("opacity reference step length") = 0.005;
69 input vec3 litdir ("view-space light direction") = [-1,-2,-1];
70 input vec3 phong ("Phong Ka Kd Ks Sp") = [0.1, 0.7, 0.2, 100];
71 input vec3 litwsp ("view-space light direction") = [1,1,1];
72 input vec3 camNearVsp ("color at near clipping plane") = [1,1,1];
73 input vec3 camFarVsp ("color at far clipping plane") = [1,1,1];
74 input real camUmax ("U: right") = 1;
75 input real camVmax ("V: down") = 1;
76 input real camDist ("ray sample position") = 0.0;
77 input bool camOrtho ("orthographic (not perspective)") = false;
78 input real camFar ("at-relative far clip distance") = 1000;
79 input real camNear ("at-relative near clip distance") = 1;
80 input vec3 camEye ("camera look-from point");
81 input vec3 camAt ("camera look-at point");
82 input vec3 camN ("camera up vector");
83 input real camNFar ("at-relative near clip distance");
84 input real camNFar ("at-relative far clip distance");
85 input real camFOV ("vertical field-of-view angle");
86 bool camOrtho ("orthographic (not perspective)") = false;
87 input int ileave ("image # horizontal samples");
88 input int ireaV ("image # vertical samples");
89 input real fMaskTh ("feature mask threshold") = 0;
90 input real fStrTh ("feature strength threshold");
91 input real thick ("approximate thickness of feature") = 0.5;
92 input real maxAlpha ("maximum opacity of feature") = 0;
93 input real alphaFix ("transparency beyond alpha") = 0;
94 input real refStep ("opacity reference step length") = 0.005;
95 input real camStep ("ray-step size");
96 input int fMaskTh ("feature mask threshold") = 0;
97 input real fStrTh ("feature strength threshold");
98 input real thick ("approximate thickness of feature") = 0.5;
99 input real maxAlpha ("maximum opacity of feature") = 0;
100 input real alphaFix ("transparency beyond alpha") = 0;
101 input real refStep ("opacity reference step length") = 0.005;
102 input real camStep ("early ray stopping transparency") = 0.005;
103 input real thick ("approximate thickness of feature") = 0.5;
104 input real maxAlpha ("maximum opacity of feature") = 0;
105 input real alphaFix ("transparency beyond alpha") = 0;
106 input real refStep ("opacity reference step length") = 0.005;
107 input real camStep ("early ray stopping transparency") = 0.005;
108 input real thick ("approximate thickness of feature") = 0.5;
109 input real maxAlpha ("maximum opacity of feature") = 0;
110 input real alphaFix ("transparency beyond alpha") = 0;
111 input real refStep ("opacity reference step length") = 0.005;
112 input real camStep ("early ray stopping transparency") = 0.005;
113 input real thick ("approximate thickness of feature") = 0.5;
114 input real maxAlpha ("maximum opacity of feature") = 0;
115 input real alphaFix ("transparency beyond alpha") = 0;
116 input real refStep ("opacity reference step length") = 0.005;
117 input real camStep ("early ray stopping transparency") = 0.005;
118 input real thick ("approximate thickness of feature") = 0.5;
119 input real maxAlpha ("maximum opacity of feature") = 0;
120 input real alphaFix ("transparency beyond alpha") = 0;
121 input real refStep ("opacity reference step length") = 0.005;
122 input real camStep ("early ray stopping transparency") = 0.005;
123 initially [raycast(ui, vi) = true];
124 | vi in 0..iresV-1, ui in 0..iresU-1 ];
```

The renderer is made specific to isosurface with the feature step \( f\text{step} \) (line 32) and feature strength \( f\text{strength} \) (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-variante data, rather than directly creating \( F \) from the data as in line 27 above. The feature mask function \( f\text{Mask} \) (line 35) described in Sec. 3.3, offers additional tunable control over what parts of a feature are worth seeing, and the test function \( f\text{Test} \) (line 36) is available as a further criterion for feature membership. These are used in the \( \text{postTest} \) 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.

```c
input real fSThresh("Feature strength threshold");
input real fMaskTh("Feature mask threshold") = 0;
input real fBias("Bias in feature strength computing") = 0.0;
input real targetInterParticleDistance; /* tsep is the length only or speed variable with data spatial
input unity every-else measures space in units of tsep */

input real minBirthDist ("Min allowed birth distance (> 0.7535)") = 0.75;
input real travMax ("Max allowed travel to or on feature") = 10;
input int nMax ("Max allowed # feature steps") = 20;

// These next three control the Gradient Descend in Energy
input input vector gDataTest("Scaling in sufficient decrease test") = 0.5;
input input vector gDataBack ("How to scale stepsize for backtrack") = 0.5;
input input vector gDataOpportunisticStepSizeIncrease = 1.2;

input input vector fSeps ("Conv. thresh. on feature step size");
input input vector geoSeps ("Conv. thresh. on system geometry") = 0.1;
input input vector vNetSeps ("Conv. thresh. on point movement") = 0.01;
input input vector rPcSeps ("Conv. thresh. on recent pop. changes") = 0.01;

input input vector phiLimit ("motion limit before PC") = 5;
input input vector isoVal ("Which isosurface to sample") = 0;

input input vector tMin ("Min allowed birth distance") = 0.75;
input input vector tMax ("Max allowed travel to or on feature") = 10;
input int nMax ("Max allowed # feature steps") = 20;

function vec3 fSThresh();
function real fMaskTh();
function real fBias();
function real targetInterParticleDistance; /* tsep is the length only or speed variable with data spatial
function unity every-else measures space in units of tsep */

function real minBirthDist ("Min allowed birth distance (> 0.7535)") = 0.75;
function real travMax ("Max allowed travel to or on feature") = 10;
function int nMax ("Max allowed # feature steps") = 20;

// These next three control the Gradient Descend in Energy
function input vector gDataTest("Scaling in sufficient decrease test") = 0.5;
function input vector gDataBack ("How to scale stepsize for backtrack") = 0.5;
function input vector gDataOpportunisticStepSizeIncrease = 1.2;

function input vector fSeps ("Conv. thresh. on feature step size");
function input vector geoSeps ("Conv. thresh. on system geometry") = 0.1;
function input vector vNetSeps ("Conv. thresh. on point movement") = 0.01;
function input vector rPcSeps ("Conv. thresh. on recent pop. changes") = 0.01;

function input vector phiLimit ("motion limit before PC") = 5;
function input vector isoVal ("Which isosurface to sample") = 0;

input input vector tMin ("Min allowed birth distance") = 0.75;
input input vector tMax ("Max allowed travel to or on feature") = 10;
input int nMax ("Max allowed # feature steps") = 20;
```

1. **input real fSThresh("Feature strength threshold");**
2. **input real fMaskTh("Feature mask threshold") = 0;**
3. **input real fBias("Bias in feature strength computing") = 0.0;**
4. **input real targetInterParticleDistance; /* tsep is the length only or speed variable with data spatial
input unity every-else measures space in units of tsep */**
5. **input real minBirthDist ("Min allowed birth distance (> 0.7535)") = 0.75;**
6. **input real travMax ("Max allowed travel to or on feature") = 10;**
7. **input int nMax ("Max allowed # feature steps") = 20;**
8. **// These next three control the Gradient Descend in Energy
input input vector gDataTest("Scaling in sufficient decrease test") = 0.5;**
9. **input input vector gDataBack ("How to scale stepsize for backtrack") = 0.5;**
10. **input input vector gDataOpportunisticStepSizeIncrease = 1.2;**
11. **input input vector fSeps ("Conv. thresh. on feature step size");**
12. **input input vector geoSeps ("Conv. thresh. on system geometry") = 0.1;**
13. **input input vector vNetSeps ("Conv. thresh. on point movement") = 0.01;**
14. **input input vector rPcSeps ("Conv. thresh. on recent pop. changes") = 0.01;**
15. **input input vector phiLimit ("motion limit before PC") = 5;**
16. **input input vector isoVal ("Which isosurface to sample") = 0;**
17. **input input vector tMin ("Min allowed birth distance") = 0.75;**
18. **input input vector tMax ("Max allowed travel to or on feature") = 10;**
19. **input int nMax ("Max allowed # feature steps") = 20;**

---

G. Kindlmann et al. / Rendering & Extracting Extremal Features

© 2018 The Author(s)

Computer Graphics Forum © 2018 The Eurographics Association and John Wiley & Sons Ltd.
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 features 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 \( \phi(r) \), implemented as \( \phi_1 \) (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 \( \phi(r) \)](image)

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 (ener on line 75) and force (frc on line 76) are defined as they were in simple Fig. 4

G. Kindlmann et al. / Rendering & Extracting Extremal Features
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 pOIter (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 a new 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 \( m_{\text{min}} = 6 \) 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 \( [nn_{\text{min}}, nn_{\text{max}}] \). 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.

\[
\frac{D}{S} = \sqrt{\frac{H^2}{S^2} - \frac{2H}{S} + \frac{5}{4}}
= \sqrt{\frac{1}{\sqrt{1 + \frac{1}{m} (5 + \sqrt{5})}}}
\approx 0.735085
\]

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 language 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 (PVO) [PR09].

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

\[
(a | b)(x) \Leftrightarrow P(x) = \frac{a(x)}{|a(x)|} \cdot \frac{b(x)}{|b(x)|} = \pm 1
\]

(24)

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

We modified the Diderot compiler to learn expressions for these derivatives, by printing \(\text{SBT}X\)or Unicode formatings of the intermediate representation. Starting with a minimal program to evaluate once the gradient of the PVO:

```plaintext
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)[1] P = (a / |a|) \parallel (b / |b|); // the PVO
7
8 strand f(int i) {
9   output tensor[3] r = \n(x) \parallel (0,0,0);
10   update {
11     stabilize;
12   }
13 }
14 initially [ f(i) | i in 0..0 ];
```

Our modified compiler generated:

\[
\frac{\partial (a \parallel b)}{\partial (b \parallel a)} + \frac{\partial (b \parallel a)}{\partial (a \parallel b)}
\]

(25)

\[
-\frac{\partial (a \parallel b)}{\partial (a \parallel b)} + \frac{\partial (b \parallel a)}{\partial (b \parallel a)}
\]

(26)

which we manually post-processed to find:

\[
\nabla P = \frac{a \cdot \nabla b + b \cdot \nabla a - a \cdot b}{|a||b|} (a \parallel b)
\]

(27)

We were not previously familiar with this expression of \(\nabla P\), which (to first order) points towards (or away from) where \(a\) and \(b\) are parallel. Terms like \(a \cdot \nabla b\) are the Jacobian \(\nabla \cdot b\) of \(b\), contracted on the left by \(a\), which can be thought of as a sum over the rows of \(\nabla b\), weighted by the components of \(a\). The \(\nabla P\) expression could also be derived by hand, but it was a nearly automatic side-effect of our modified Diderot compiler. The expression for \(\nabla P\) is symmetric in switching \(a\) and \(b\), 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 \(\varepsilon\) from \(x\) such that \(x + \varepsilon\) satisfies \(a \parallel b\) (c.f. (29) in [GP09]):

\[
q + \frac{1 - b \cdot b}{b \cdot b} (\nabla \cdot a - \varepsilon \nabla b) \varepsilon = 0
\]

(28)

where

\[
q = \left(1 - \frac{b \cdot b}{b \cdot b}\right) a
\]

(29)

is the component of a orthogonal to \(b\). The authors then describe how \(\varepsilon\) may then be 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 \(a\) and \(b\).

We were curious if our modified Diderot compiler could produce a human-readable expression for the Hessian of \(P(x) = \frac{a(x)}{|a(x)|} \parallel \frac{b(x)}{|b(x)|}\), which is inverted as to compute, via (12), the feature step of our approach. By changing line 9 in the program above to include \(e = \nabla \cdot \nabla P([0,0,0])\); our modified compiler generated a lengthy expression:
With some manual post-processing (factoring common terms and regrouping), we develop an expression for the Hessian of $P$:

$$
\nabla \otimes \nabla P = \frac{(\nabla \otimes b)^T \cdot \nabla \otimes a + (\nabla \otimes a)^T \cdot \nabla \otimes b + a \cdot \nabla \otimes \nabla \otimes b + b \cdot \nabla \otimes \nabla \otimes a}{a \cdot a}
$$

Review of this expression reveals that it too is symmetric in switching $a$ and $b$. $\nabla \otimes \nabla \otimes a$ is the Hessian of vector field $a$, a third-order tensor that, when right multiplied by offset $e$, 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 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 (\texttt{colspan})

```cpp
if (c != c1) { if (c > c1) which = 2; }
else if (1 == which) { ret += c2; return normalize(result) if possible; }
```

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 (\texttt{edge.diderot})

```cpp
input vec3 ipos ("point positions") = load(iPos.nrrd); // the output of this program is via its many print statements,
input int tri ("triangles as triplets of point indices") = 3600; // more common informative Diderot program. Like
input real camDist ("screen point") = 1; // the sphere test implicitly depends on pos
input int triNum ("triangles as triplets of point indices") = 3600; // computing ray parameters and view-space basis
input int triNum ("triangles as triplets of point indices") = 3600; // now with camAtShift
input string triNum ("triangles as triplets of point indices") = 3600; // computing ray parameters and view-space basis
input string triNum ("triangles as triplets of point indices") = 3600; // now with camAtShift
```

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 \texttt{rad} given to on line 2 should be \(2/3\) of the target inter-particle distance (\texttt{ti.pd}, 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 \texttt{sphere} test (line 12) is executed efficiently.

D.3. PostScript mesh drawing (\texttt{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,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. Within 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 the \texttt{edge.diderot} above, the useful output of this program is via its many \texttt{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.

```cpp
input vec3 ipos ("position") = load(iPos.nrrd); // the output of this program is via its many print statements,
input int tri ("triangles as triplets of point indices") = 3600; // more common informative Diderot program. Like
input int triNum ("triangles as triplets of point indices") = 3600; // computing ray parameters and view-space basis
input int triNum ("triangles as triplets of point indices") = 3600; // now with camAtShift
```

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 \texttt{rad} given to on line 2 should be \(2/3\) of the target inter-particle distance (\texttt{ti.pd}, 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 \texttt{sphere} test (line 12) is executed efficiently.
true?

```c
bool snsgn = true if (feat == "FEAT-ISO") else false if (feat == "FEAT-VSF") else false if (feat == "FEAT-RLN") else true

if (!snsgn || (toeye0 >= 0 && toeye > 0))
    print("stroke1");

if ((nmw >= 0) && (nsw >= 0)) {
    print("0 0 1 0 360 arc closepath fill 1");
}
```

```c
if (snsgn)
    print("gsave");

if (printps[0] == 0)
    print("gsave 0.5 setgray");
```

```c
// convert homogeneous coords
tensor[4, 4] orthovtc = [1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1];
tensor[4, 4] vtcv = orthovtc if cmvortho else perspvtc;
tensor[4, 4] vtcC = vtcv if cmvortho else orthovtc;

// how to approximate surface "normal"?

function vec3 hom(vec3 c) = [[0, 0, 1], [0, 1, 0], [1, 0, 0]]; return ret;

} return ret;

```
vec3 pw0 = ipos[pi0];
vec3 pw1 = ipos[pi1];
vec3 pw2 = ipos[pi2];
vec3 pwn = (pw0 + pw1 + pw2)/3;
vec3 pwn = snorm(pwn);
vec3 pwm = (pw0 + pw1 + pw2)/3;
vec3 toeye = normalize(canEye - 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 wz = min(min(ps0[2], ps1[2]), ps2[2]);
  if (-1 <= wz && wz <= 1 && trigray <= 1) {
    print(wz, " 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("* pop ");
  print("grestore grestore\n");
  stabilize;
}
initially | draw(ii) | ii in 0 .. (pntNum+edgNum+triNum)-1 |;