Point Movement in a DSL for Higher-Order FEM Visualization

Teodoro Collin\textsuperscript{1} \hspace{1cm} Charisee Chiu\textsuperscript{2} \hspace{1cm} L. Ridgway Scott\textsuperscript{1} \hspace{1cm} John Reppy\textsuperscript{1} \hspace{1cm} Gordon Kindlmann\textsuperscript{1}

\textsuperscript{1}Department of Computer Science, University of Chicago\textsuperscript{*} \hspace{1cm} \textsuperscript{2}Galois, Inc.\textsuperscript{†}

\textbf{Abstract}

Scientific visualization tools tend to be flexible in some ways (e.g., for exploring isovales) while restricted in other ways, such as working only on regular grids, or only on unstructured meshes (as used in the finite element method, FEM). Our work seeks to expose the common structure of visualization methods, apart from the specifics of how the fields being visualized are formed. Recognizing that previous approaches to FEM visualization depend on efficiently updating computed positions within a mesh, we took an existing visualization domain-specific language, and added a mesh position type and associated arithmetic operators. These are orthogonal to the visualization method itself, so existing programs for visualizing regular grid data work, with minimal changes, on higher-order FEM data. We reproduce the efficiency gains of an earlier \textit{guided search} method of mesh position update for computing streamlines, and we demonstrate a novel ability to uniformly sample ridge surfaces of higher-order FEM solutions defined on curved meshes.

\textbf{Index Terms:} Software and its engineering—Software notations and tools—Context specific languages—DSLs

\section{Introduction}

Novel methods of high-performance and scalable scientific visualization typically support interactively exploring various parameters (e.g., volume rendering transfer functions, or streamline seedpoints), while constraining the form of data being visualized. That is, tools for visualizing large biomedical imaging volumes are sensibly specialized for the regular grids that such data is acquired on, just as fluid flow visualization tools are specialized for the finite element method (FEM) meshes on which those phenomena are simulated. Yet from the high-level mathematical standpoint of either characterizing existing visualization methods, or exploring the value of new ones, the specialization of tools to data forms is unfortunate: volume rendering is a kind of integration, and streamlines are solutions to ODEs, regardless of how exactly scalar or vector fields are defined on a grid or mesh. Visualization research may benefit from systems that can take a high-level specifications of a visualization method, and a separate description of how data and fields are formed, and then compile programs that both run efficiently on the given data and support exploration of the relevant parameter spaces.

Domain-specific languages (DSLs) for scientific visualization partially address this need by specializing for a class of algorithms and one form of data: regular grids \cite{1,10,17,22,23,35}. In their own ways, these languages work to separate the legible expression of visualization algorithms from technical details of data access or parallel computing. Our current work, however, explores whether a scientific visualization DSL can also be general with respect to data form, so that a program that works efficiently for data on a regular grid can, with minimal changes, also work on an unstructured mesh. Particularly challenging are higher-order finite element simulations, which use higher-order polynomials in both the geometry of mesh elements (so they can be curved) and the form of solutions within elements (beyond affine functions), since this increases the implementation complexity of mere field evaluation.

We present preliminary work on extending the open-source compiler for a scientific visualization DSL \cite{1,19,20}, previously limited to regular grids, to also work on higher-order FEM data. Our long-term goal is to connect previous FEM visualization methods \cite{8,15,22,27} by simplifying how they can be expressed and combined in working code. Our current focus is just two kinds of visualizations, both involving computations on a discrete set of points: streamlines \cite{11}, and particle systems sampling surface features \cite{25}. The inner loops of both methods share an essential element, point movement, i.e., incrementing a position within a higher-order FEM domain by some update vector. We seek a programming language that allows clearly legible implementations of point movement (as a specification of an aspect of the data form), and of the visualization method itself, to be combined in a single program.

Our main contribution approaches this by demonstrating how adding a type for representing FEM mesh positions to a DSL, and overloading operators on that type, simplifies implementing visualization methods that rely on point movement. A supporting contribution, ridge surface extraction in a curved finite element mesh by a particle system, uses the main contribution, and also suggests how other new visualizations could be created by combining general methods with specializations to data form. We hope our work (which itself will be made open-source available) can eventually lower the implementation complexity of mere field evaluation, as well as help extend standard visualization algorithms to other more general domains (e.g., manifolds) in which points and vectors have distinct significance.

\section{Finite Element Method (FEM) Background}

A brief explanation of a simplified and typical use of FEM will support a description of our work. For a solution to a partial differential equation (PDE) $u : \Omega \rightarrow \mathbb{R}^n$, FEM uses a finite dimensional vector space of functions (function space) $V$ to find an approximate solution, $u_V \in V$, to $u$. The space $V$ is created by discretizing the world-space domain $\Omega$ into a collection of disjoint cells $\{K_i\}$, defining a function space $P_i$ on each $K_i$, and then combining all the $P_i$, e.g., $v \in V$ if and only if $v_{|K_i} \in P_i$ for all element indices $i$ \cite{6}. The per-element function space is typically $P_i = \{p \circ T_i^{-1} | p \in P\}$, where $P$ is a finite dimensional function space on a convex polytope $K$ (the reference cell), each $T_i$ is an injective $C^\infty$ mapping from $K$ to world-space, and both $P$ and $T_i$ are polynomial. Consequently, finite element solvers do not need to explicitly represent the $P_i$ (or compute $T_i^{-1}$), and can compute all quantities on the reference cell $K$ \cite{14,16,24,33}.

Unfortunately, visualization naturally works in world-space $\Omega$. Within some $K_i \subset \Omega$, the PDE solution being visualized $u_V$ will be represented in a chosen basis $\{p_j\}$ for $P$ as

$$u_V(x)_{|K_i} = \sum_j c_j p_j(T_i^{-1}(x)).$$

(1)

Using higher-order FEM, with non-linear $T_i$, increases the computational cost for a naive visualization algorithm to traverse just a single cell $i$, since each of the many evaluations of $T_i^{-1}$ in \cite{14} require

*e-mails: \{teocollin,ridg,jhr,glk\}@cs.uchicago.edu
*e-mail: chiw@galois.com
multiple Newton iterations. Moreover, as the visualization traverses world-space, for each point \( x \in \Omega \) it needs to compute \( T_i^{-1} \) for many different cells \( i \) in order to find the \( i \) for which \( x \in T_i(K) \) [30]. Visualization algorithms specialized to finite elements avoid the cost of \( T_i^{-1} \) by replacing, when possible, world-space evaluation of the approximate solution \( u_T \) via (1) with

\[
f_i(x) := \sum_j c_j p_j(x), \tag{2}
\]

where \( f_i : K \to \mathbb{R}^n \) is the evaluation of \( u_T \) on the reference cell with respect to element \( i \).

3 RELATED WORK

Many tools for visualizing FEM solutions (including ParaView [3], Gmsh [15], and GLVis [16]) use tessellation, i.e., approximating one higher-order element with multiple smaller affine elements [30]. This is good for simple visualizations (e.g. coloring mapping \( u_T \)), but more problematic for more complicated ones, such as volume rendering or those that require higher order derivatives [30]. The tessellation framework of Schroeder et al. is in principle general and accurate with respect to visualization method [38], but we are unaware of its application beyond isosurfaces and streamlines.

Our work follows a different strategy, advanced by Nelson et al., which directly visualizes elements, without tessellation. Via algorithms that directly manipulate \( u_T, f_i, K \) and \( T_i \), these authors create fast and accurate methods for ray-tracing isosurfaces, cut surfaces, and volume rendering [27–29]. They also combine methods into ElVis, a GPU-based interactive GUI application, which offers some generality over data forms via a plugin architecture that supports a small set of visualization algorithms [30]. Our DSL, however, allows more room to explore implementation variation within or between data forms, at the expense of lower computational performance relative to hand-written low-level code.

A variety of other previous work investigates FEM visualization under various accuracy, expression, and performance constraints, as surveyed by Nelson et al. [30]. Some work focused on fast and accurate visualizations of curved quadratic and cubic elements [40] while other work achieved interactive volume rendering of higher order elements [40]. Later sections will describe in more detail the work of Coppola et al. [11] and Meyer et al. [25], which is most central for our current work. There is no prior work extracting ridge surfaces from FEM data, but Pagot et al. find ridge lines on affine meshes via PVO and new seed finding and streamline routines [51]. Jallepalli et al.’s smoothing of finite element data could usefully complement the visualization methods we target [18].

We also build on related work with DSLs. Being specific to some domain of algorithms, DSLs trade reduced flexibility of the language for (in principle) higher human productivity of writing new programs within that domain [24]. For our purposes we merely note FEM-related DSLs for formulating and solving PDEs [12,13,34] as well as DSLs for processing and visualizing image and volume data [6,10,17,22,23,26,33,35]. This list does not fairly describe the sophisticated approaches to high-performance computing [23] and computational scheduling [26,33]. We build on Diderot, a visualization DSL limited to regular grids [6,19,20], but distinguished by offering the mathematical abstraction of a \( C^k \) tensor field. Our current work extends how Diderot fields are defined to include FEM, so that existing Diderot programs can be used with minimal changes, while introducing a new abstraction, a mesh position, which supports the convenient expression of previous methods of moving through the geometry of a curved FEM mesh [11,25].

4 METHODS

4.1 Point Movement via Guided Search

Many scientific visualization algorithms enjoy spatial coherence: field evaluation at (world-space) position \( x \) will likely be followed by evaluation at a nearby \( x + v \). As noted in [32] for simple methods, the computational expense (from finding the cell containing \( x \) and finding \( T_i^{-1}(x) \)) of naively evaluating \( u_T(x) \) via (1) might be avoided by evaluations in reference space via (2), and then forward mapping by \( T_i \). Previous work with more complex visualization methods, however, demonstrates the value of rapidly approximating \( x + v \) in a sequence of reference spaces, a technique we term point movement, so that \( u_T(x + v) \) can be found faster than via naive re-evaluation of (1). For streamlines, Coppolla et al. name their method of point movement guided search [11], while a similar method underlies the isosurfacings of particles of Meyer et al. [25].

Guided search builds on a few technical observations. First, for world-space position \( x \in T_i(K) \) and update \( v \), a first-order approximation of the updated position is \( T_i^{-1}(x + v) \approx T_i^{-1}(x) + (DT_i^{-1})^T v \). Second, the location where \( DT_i^{-1}(x + (DT_i^{-1})^T v) \) exits \( K \) can be found via geometric computations on the shape of \( K \), common in computer graphics [14]. Third, in most meshes, the mapping between reference positions in two adjacent cells is entirely determined by a simple permutation \( \sigma \) on the vertices of \( K \). Thus for \( x \in \partial K \) on the boundary of the reference cell for cells \( i \) and \( j \), the same world-space position is both \( T_j(\sigma(x)) \) and \( T_i(x) \). Combining these ideas together yields the guided search algorithm illustrated in Fig. 1.

We make two observations about the context and implementation of guided search. Guided search has only been described as a part of a specific visualization ingredient: Coppolla et al. presents guided search as a sub-step of RK4 integration [11]. In fact, it can be separated from any particular numerical or visualization method and framed as a method to update positions by a vector, a definition in affine geometry [39]. Second, guided search is complicated enough to warrant exploring the speed and accuracy of possible variants. For example, Coppola et al. also describe error-checked guided search, wherein the search defaults to the naive scheme to locate \( x + v \) if \( ||T_i(T_i^{-1}(x) + ( DT_i^{-1} )^T v) - (x + v)|| \) exceeds some threshold. The same considerations of orthogonality and legibility that motivate creating DSLs also suggest clearly expressing the point movement method within the language.

4.2 FEM Data, Position types, and Overloading

To demonstrate point movement within a mesh as a programmable and orthogonal aspect of a FEM visualization algorithm, we augment an existing scientific visualization DSL with a new position type, overloaded operators on positions, and the ability to input FEM data. We chose the Diderot language because it already simplifies implementing streamlines and particle systems on regular grids [19, 20], and because its consistent use of a field abstraction facilitates introducing FEM solutions as a new underlying data form.

Space here permits a high level summary of the language changes

![Figure 1: Illustration of guided search to move a towards b, with two world-space cells (left) and two copies of reference space (center and right), each with transforms \( T_1 \) and \( T_2 \). Colored shapes at cell vertices clarify how reference spaces connect. At \( a \), the velocity \( v = b - a \) is transformed by \( DT_1^{-1} \) to give a reference space velocity \( DT_1^{-1}v \), along which traversal begins. At cell boundary \( m_1 \), the permutation \( \sigma \) between reference vertices determines the start \( m_2 = \sigma(m_1) \) of the next cell traversal, now along \( DT_1^{-1} \) and \( m_2 \). More cells may be encountered, until the computed path (shown as a dashed line) has run for time \( t = 1 \), ending at \( m_1 \). The point \( T_2( m_1 ) \) approximates \( b \).](image-url)
Figure 2: An overloaded "+" operator implements a minimal version of guided search along with minimal context. Lines 1-5 declare the necessary inputs. Lines 7-9 set up the search. Lines 10-29 are the main body. At each iteration, lines 11 abd 12 transform the change in position via the position’s cell’s transform. Lines 14 and 15 return a new position if the transformed velocity does not take the position outside the cell. Otherwise, lines 17-28 find the time of intersection (line 17), check that the intersection makes sense (line 19), find the position of this intersection on the next cell (line 22), check if this is the last step (line 23), and continue on (line 26). If the loop terminates, line 29 defaults to the naive scheme.

```
1 input int timeSteps=12;
2 const int dim = mesh_t.dim; // (this works in 2 or 3 dimensions) 17
3 input real timeEps = 0.000001; 18
4 input mesh_t mesh; 19
5 refCell(mesh_t) K = mesh.refcell; 20
6 overload position(mesh_t) + (position(mesh_t) x, tensor[dim] delta); 21
7 if (x.isValid) (return(x)); 22
8 real time = 1; 23
9 position(mesh_t) cmp = x; 24
10 foreach (int i in 0 .. timeSteps){ 25
11 tensor[dim, dim] J = (cmp.mc.transform)cmp.refPosPos); 26
12 tensor[dim] refDelta = J * delta; // reference velocity 27
13 tensor[dim] nPos = cmp.refPos + time*refDelta; 28
14 if (K.inside(nPos)) { 29
15 return(cmp.mc.meshPos(nPos)); 30
16 // spent too much time; use naive scheme 31
} 32
// invalid direction , use naive scheme 33
// we left the reference cell; compute when we left 34
real eTime = K.exit(cmp, normalize(refDelta)); 35
// decrement time remaining if (eTime == -1) [ // invalid direction, use naive scheme 36
return(mesh.findPos(x.worldPos() + delta)); 37
} 38
// else find the exit location in the next cell 39
position(mesh_t) cmp = K.exit(cmp, normalize(refDelta)); 40
if (cmp.isValid || time < timeEps){ 41
return(cmp); 42
} 43
// run out of time 44
}
```

Figure 3: Streamlines in this synthetic vector field in a curved mesh should be helices of constant radius. Three schemes for updating position during integration are seeded at the same location (right inset), but guided search (green) diverges by the end (left inset). Guided search with error checking (blue) very closely follows the accurate and more expensive naive method (orange).

5 RESULTS

Our results all use curved meshes with cubic \( T_i \) transforms. Additional software was used to create meshes (gmsh [15]), finite element data (Firedrake [32]), and renderings (ParaView [13]). The Diderot code can be found in the supplementary materials.

To test streamlines, we created a curved mesh between two concentric cylinders. We then interpolated \( f(x, y, z) = (y, -x, 0.1) \) onto a function space specified by the mesh and quadratic \( P \). An RK2 streamline program with guided search produces the green path in Fig. 5. Also shown is an orange path produced via the more expensive naive scheme, which shows the accuracy of the blue path computed with the error-checked guided search noted in [11]. This result is consistent with Coppola et al.’s accuracy analysis of these schemes [11]: standard guided search errs in ways that error-checked guided search avoids.

Coppola et al. also analyze the performance of guided search. Our preliminary results in Table 1 reproduce their finding that error-checked guided search runs 2–10 times faster than the naive approach; this is notable considering that our code is in a new high-level DSL. We are also encouraged by this speed-up because it justifies the compiler and language effort of [44] and facilitates future work on exploring new point movement techniques independently of visualization methods.
Meyer et al. pioneered isosurfacing via particle systems on curved geometries [25]. It took us a few hours to adapt an 80-line minimalist isosurface sampling Diderot program [20] to produce Fig. 5 showing a sampling of the isocontour \( x^2 + y^2 + z^2 = 1 \) in a cylindrical mesh with hexic \( P \). Our result lacks curvature-adaptive sampling [25], but it shows the viability of our approach.

We also sample a ridge surface of the function \( f(x,y,z) = z^2 \sin(x^2 + y^2 + z^2) \), inspired by Eberly’s consideration of ridges in fluid flow (c.f. Fig. 6.49 in [13]). We created a curved mesh between two concentric spheres, and approximated \( f \) in a function space given by the mesh and hexic \( P \). Since \( f \) is non-polynomial, the approximation is at most \( C^0 \) continuous across cell boundaries, which could create discontinuities in the ridge surface itself. A 300-line Diderot program for sampling general features with particle systems was adapted as described in [4, 2] and the results in Fig. 5 were found after experimenting with parameters (feature strength threshold of 24, feature bias of 0.1). We spent more time creating the example function and mesh than we did writing and using the program. Therefore, we feel that the most interesting aspect of this result is in the DSL design. The conceptual orthogonality of guided search and particle system evolution was manifested as a clean separation in the code between the implementation of those two methods, such that the guided search code remained unchanged as the particle system code was changed from isocontours to ridge surfaces.

### 6 Discussion, Conclusions, and Future Work

Our methods and results demonstrate that point movement and visualization algorithms can be orthogonal and composable. We have shown how conceptually orthogonal algorithmic components can be cleanly expressed as separate pieces of code in a DSL. The ridge surface example additionally shows that this separation extends in concept and in code to particle systems for a novel visualization target for FEM data. We hope these results convince readers of the potential of high-level DSLs to create new visualization programs by combining two separate specifications: one of the data form (regular grid or FEM data), and one of the core visualization algorithm.

Directions of ongoing and future work are organized around Diderot, guided search, and the expression of visualization algorithms in general. With respect to Diderot, we hope to add other data forms beyond regular grids and finite element data such as Riemannian manifolds, where position movement is given by solving a variational problem to find a geodesic. With respect to guided search, we wish to augment it with information from run-time or compile-time, and we wish to explore its application to visualization methods that are less sensitive to errors in position location (i.e., volume rendering or predictor-corrector schemes used in PVO line tracing [21, 31]). Finally, we suspect that many existing implementations of visualization algorithms currently specialized to a particular data form may contain ideas that are as orthogonal and composable as guided search is for FEM data. We hope to foster further research by uncovering those ideas and exploring how they can be combined into new visualization algorithms, implemented in idiomatic and re-usable code.

**Acknowledgments**

This work was supported by NSF grant CCF-1564298.
7 SUPPLEMENTARY MATERIALS FOR POINT MOVEMENT IN A DSL FOR HIGHER-ORDER FEM VISUALIZATION

8 SUMMARY OF SUPPLEMENTAL MATERIALS

In this document, we will include the commented code that is used by “Point Movement in a DSL for Higher-Order FEM Visualization.” Each section will provide Diderot snippets that could be used to produce the figures in the paper. These snippets are complete in the sense that they are either full programs in their own right or could easily be combined with another snippet featured here to make a full program. In addition, each section will provide some brief commentary on the code, highlighting both the new ideas in the code not evident before and the work required to create these snippets from older Diderot programs. The snippets included below are:

1. A streamline program that uses guided search.
2. An overloaded function for the error checked guided search scheme.
3. An overloaded function for the naive scheme.
4. A small particle system used to create isosurfaces.
5. A larger particle system used to create ridge surfaces.

Finally, we note that the figure captions provide information about parameters that we used to create our figures if they are unstated in the paper.

9 STREAMLINES

We provide line by line commentary on Figure 6 which features a complete guided search streamline program. Line 1 declares the version of the compiler that we use, which is 3.0. Lines 2 through 7 declare the mesh, function space (V), and function types (uV) that the program uses. The JSON files used in these declarations will be documented further in future work, but we note that we automatically generate them for Firedrake’s meshes, spaces, and functions.

Lines 10 through 12 take inputs for the mesh, space, and function. Line 13 provides the reference cell from the mesh. Lines 18 through 44 are featured in the paper as the guided search algorithm. Lines 46 through 54 supply an auxiliary function, nV, to unpack positions and evaluate the field f1 on the reference cell; this process consists in taking out a mesh cell, reference position, and then using the information about uV, the data, to get a field, which is sampled at the reference position. The auxiliary function checks the validity of the position and allows for some sort of border control. Lines 55 through 57 simply control the streamline algorithm, RK2. Lines 58 through 73 are the streamline algorithm with some modifications to use positions: First, line 64 takes the input vector to the strand, line, and converts it to a position. Second, lines 66 and 69 provide border control by checking for the validity of the positions (in older Diderot versions, this was done via an inside function.) Finally, line 67 records the world position of a point on the streamline. The core RK2 algorithm is legible in lines 68 and 70, which use the function nV to sample a field at a position as in the older Diderot streamlines program.

We now roughly measure the changes required to turn a vanilla Diderot streamline program to this program. A standard Diderot streamline program would live in lines 46 through 73 after the addition of image inputs and field declarations. The changes in the function, nV, defined from 46 to 53, are changes in the field evaluation and represent an additional 5 to 10 lines from the original program. The strand definition is changed at lines 64 (type change and conversion), line 66 for checking the validity, line 67 for recording the world position, line 68 for a type change, and line 69 for another validity check. Thus, the total changes to the streamline program amount to about 15 lines of code besides the addition of the FEM inputs and guided search (line 1 through line 44).

In Figure 7 we show the overloaded position operator with an error checking functionality. An error max parameter is required as a new input on line 1 and we now provide commentary on its usage i.e the additions present in the error checked guided search. The error parameter is used on lines 14 and 33 to check if the computed reference position corresponds to a world space position that is close to the correct world space position i.e the world space position of a naive position update. If on either line 14 or 33, the condition is met, the program continues as in the previous figure, but otherwise, the new program tries to recover somehow. On line 14, the code is finishing inside the reference cell, and, therefore, can use the current cell to check if the correct world space position corresponds to a reference position in the current cell. If the current cell does contain the correct reference position, the correct reference space position is computed via T−1, but if the current cell does contain the correct reference position, the naive scheme is used. On line 33, the position is currently on a boundary so use of T−1 is inappropriate; thus, the naive scheme is used immediately on line 34.

Finally, in Figure 8 we provide an overload for the naive scheme. This is provided for clarity and does not require comment.

10 PARTICLES

In this section, we will provide the complete programs used to make the particle system figures. The position addition overload in these programs does not differ from Figure 6 nor do the creation of types or finite element inputs. In this section, we will focus on the changes to the core particle systems programs; besides the addition of the overloaded functions and finite element inputs, how many lines were changed in porting these programs? How much did the main loop change? We stress that one should examine the prior work on Diderot to understand the full particle system programs and that here we mainly seek to point out the limited extent to which FEM versions differ from the original programs.

We first consider the shorter particle systems program used to create the isosurface in the paper, displayed in Figure 9. Lines 1 through 45 are basically identical to those for guided search in Figure 6. Line 46 through 52 implement position subtraction, which is a standard position operation, via taking world space differences if both positions are valid, and otherwise returning zero. Lines 54 though 168 implement a small particle system for an isosurface. Lines 54 through 58 are parameters to the system as in the original program. Lines 61 through 89 implement feature strength, feature step, and feature perpendicular functions that sample positions; these functions are considered inputs to the particle system code. These are different from the original versions of these functions, but they are trivially the same only sampling through the reference space via unpacking the position, accessing the current cells, and acquiring the transformed reference fields described in the paper. In terms of lines of code, each differs by 5 to 10 lines from the original, leading to an additional roughly 10 to 20 lines, but, we note these functions live outside the core particle system code. The core of the particle systems program lies in lines 89 through 168. We observe that this is basically identical to the original program except for 7 lines (lines 96, 97, 105, 128, 144, 169, 166). Each line has a comment explaining the change relative to the old program, but we note that 4 lines differ by type, two lines use the validity method of a position to do border control, and only lines 163 and 166 implements new functionality. In short, for a core loop of 72 lines, only 2 to 4 were added or changed non-trivially. In short, the core logic of the small particle system program is basically unchanged in the conversion to use guided search modulo the addition of the overloads on positions and the specification of finite element data. Examining our analysis, we see that besides the FEM inputs and position overloads, the program features at most 30 lines of additions or changes.
This result is repeated with the larger particle system program, which is featured for ridge surfaces in Figure 10 and Figure 11. Combining these figures yields the full program. In the first part of the code, Figure 10 we find the particle system controls, guided search controls, and particle system auxiliary functions. Lines 1 through 53 provide FEM inputs and guided search, as in the previous examples. Lines 54 through 75 provide the particle system controls. Lines 78 through 126 provide the feature step, perpendicular, strength, and test functions that these particle systems take as inputs. As before, these are the same as their vanilla versions, but they take positions, unpack them, and sample from the reference cell. We don’t provide additional commentary on these functions as the conceptual extent to which they differ from the original Diderot programs is the same as in the previous particle system. The step, perpendicular, and strength functions require a few additional lines (at most 5 each) where as the mask, test, and posTest functions change by 1 line, 1 and 2 lines respectively. Thus lines 78 through 126 feature at most 20 additions or changes to the code. Lines 127 through 180 are almost identical to the original code, but the v3rand and genID functions (lines 175 and 166) need to use positions instead of vectors, creating another 4 changes. Moving on to the second part of the large particle system, Figure 11 we find the core particle system code. In Figure 11 all comments have been removed except those that indicate that a change has been made from the original program. We find that there are no more than 7 lines of changes and that only the last two lines, which check that all positions are valid before allowing convergence, are substantial changes. Combining this with our analysis of the first part and, as before, discounting the added overloads and FEM inputs, the total changes between this program and its regular grid version amount to fewer than 35 lines of code.

---

```cpp
// Version 3.8
// mesh specification:
#version 3.0
femFunction{fns_t} func_t = file("evalProg.json");
functionSpace{mesh_t}[dim] fns_t = file("evalProg.json");
type femFunction(fns_t func_t = file("evalProg.json");

// space (V) specification:
const int dim = mesh_t.dims; // dim of mesh

// input mesh
input mesh_t mesh; //input mesh
input fns_t space = fns_t(mesh); //input space
input func_t data = func_t(space); //input u(V)
input real position = real(0..timeSteps);
input int overload position(mesh_t) + (position(mesh_t). x, tensor<dim> worldDelta) { (x.isValid) { step+=1; return(x); } if (x.isInside(nPos)) { return(cmp.mc.meshPos(nPos)); } else { tensor<dim> nRefDelta = normalize(refDelta); real eTime = cmp.mc.transform(cmp.refPos); time+=eTime / refDelta; if (eTime == -1) { return(mesh.findPos(x.worldPos() + worldDelta)); } } return(mesh.findPos(x.worldPos() + worldDelta)); } return nmp = K.exitPos(cmp, nRefDelta); if (nmp.isValid) { stream = {mesh_t}(mesh.findPos(startPos)); } else return(zeros[dim]);)

//find position corresponding to global position of startPos
position(mesh_t) cPos = mesh.findPos(startPos);
update { //trace if position is valid
if(cPos.isValid() || step == stepMax) { stabilize(); } stream = stream0(cPos.worldPos()); } position(mesh_t) intermed = cPos + 0.5 * stepSize * nV(cPos); if(!intermed.isValid) { stabilize(); } validity of substep cPos = cPos + stepSize * nV(intermed); step++; }

//create_collection {line(x) | x in startPoints};
```

---

Figure 6: A complete RK2 streamline program that uses guided search
input real errorMax = 1.0;

overload position(mesh_t) + (position(mesh_t) x,
  tensor[dim] worldDelta) {
  if (!x.isValid) return(x);
  real time = 1;
  position(mesh_t) cmp = x;
  foreach (int i in 0..timeSteps ) {
    tensor[dim] iJac = inv( ∇ ⊙ (cmp.mc.transform)(cmp.refPos));
    tensor[dim] refDelta = iJac * worldDelta;
    tensor[dim] nPos = cmp.refPos + time * refDelta;
    if (K.isInside(nPos)) {
      position(mesh_t) nmp = cmp.mc.meshPos(newPos);
      //check error:
      if (cmp.worldPos() + dPos | > errorMax){
        vec3 guess = (cmp.worldPos() + time * dPos);
        if(cmp.mc.isInside(guess)){ //guess current cell
          return(cmp.mc.meshPos(cmp.mc.inverseTransform(guess));
        } else ( //default to naive scheme
          return(meshData.findPos(x.worldPos() + dPos));
        )
        return(nmp);
      } else {
        //spent too much time - use naive scheme
        return(meshData.findPos(x.worldPos() + dPos));
      }
    }
    cmp = nmp;
  }
  return(nmp);
}
Figure 9: A complete though minimal particle system that uses guided search and is aimed at isosurfaces. We note that the figure in the paper used this program with $\varepsilon_p=0.005$, $\alpha=0.5$, and $\theta=0$. 

```c
//mesh specification:
type mesh_t = file("evalProg.json");
//space (V) specification:
type functionSpace{mesh_t} fn_v_t = file("evalProg.json");
//a(V) specification:
type femFunction{fn_v_t} func_t = file("evalProg.json");

const int dim = mesh_t::dim; //dim of mesh

input mesh_t mesh; //input mesh
input fn_v_t space = fn_v(mesh); //input space
input func_t data = func_t(space); //input u(V)
refCell{mesh_t} K = mesh.refCell; //get refCell

input int timeSteps = 12; //guided search step limit
input real timeEps = 6.000000e-01; //1 - time max
output position{mesh_t} x, y; //current mesh pos
output tensor{dim} delta; //current mesh vel

if (x.isValid) return (x);
real time = 1; //time
position{mesh_t} cmp = x; //current mesh pos
foreach (int i in timeSteps) {
    tensor{dim, dim} iJac = inv(V(transformRefField)(cmp.refPos));
    tensor{dim} refDelta = iJac \* worldDelta; //reference velocity
    tensor{dim} npos = cmp.refPos + refDelta - refCell;
    if (K.isInside(nPos)) {
        norm = (f.transformedRefField)(x);
        grad = (f.transformedRefField)(x);
        refDelta normalize(refDelta); //reference velocity
        real eTime = K.exit(cmp, refDelta);
        if (eTime == -1) { //invalid direction, use naive scheme
            return (mesh.findPos(x.worldPos()) + worldDelta);
        } else (found = true; testp=1);
    } else {
        force = fPerp(pos)
        pos = pos + fStep(pos);
        nmp = (newE - oldE) / (rad * rad);
        pos = pos + step;

        if (found) {
            step = fStep(pos);
            pos = pos + step;
            if (force > rad) {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            } else {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            }
        } else {
            return (newE - oldE > 0.5 * (pos - oldPos))
            pos = oldPos;
            force = fStep(pos);
            nmp = nmp + step;
        }
    }
}

output position{mesh_t} x, y; //changed the type
output tensor{dim} delta; //changed the type
output real time = 1; //time
output position{mesh_t} cmp = x; //current mesh pos
foreach (int i in timeSteps) {
    tensor{dim, dim} iJac = inv(V(transformRefField)(cmp.refPos));
    tensor{dim} refDelta = iJac \* worldDelta; //reference velocity
    tensor{dim} npos = cmp.refPos + refDelta - refCell;
    if (K.isInside(nPos)) {
        norm = (f.transformedRefField)(x);
        grad = (f.transformedRefField)(x);
        refDelta normalize(refDelta); //reference velocity
        real eTime = K.exit(cmp, refDelta);
        if (eTime == -1) { //invalid direction, use naive scheme
            return (mesh.findPos(x.worldPos()) + worldDelta);
        } else (found = true; testp=1);
    } else {
        force = fPerp(pos)
        pos = pos + fStep(pos);
        nmp = (newE - oldE) / (rad * rad);
        pos = pos + step;

        if (found) {
            step = fStep(pos);
            pos = pos + step;
            if (force > rad) {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            } else {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            }
        } else {
            return (newE - oldE > 0.5 * (pos - oldPos))
            pos = oldPos;
            force = fStep(pos);
            nmp = nmp + step;
        }
    }
}
```

Figure 9: A complete though minimal particle system that uses guided search and is aimed at isosurfaces. We note that the figure in the paper used this program with $\varepsilon_p=0.005$, $\alpha=0.5$, and $\theta=0$. 

```c
//mesh specification:
type mesh_t = file("evalProg.json");
//space (V) specification:
type functionSpace{mesh_t} fn_v_t = file("evalProg.json");
//a(V) specification:
type femFunction{fn_v_t} func_t = file("evalProg.json");

const int dim = mesh_t::dim; //dim of mesh

input mesh_t mesh; //input mesh
input fn_v_t space = fn_v(mesh); //input space
input func_t data = func_t(space); //input u(V)
refCell{mesh_t} K = mesh.refCell; //get refCell

input int timeSteps = 12; //guided search step limit
input real timeEps = 6.000000e-01; //1 - time max
output position{mesh_t} x, y; //current mesh pos
output tensor{dim} delta; //current mesh vel

if (x.isValid) return (x);
real time = 1; //time
position{mesh_t} cmp = x; //current mesh pos
foreach (int i in timeSteps) {
    tensor{dim, dim} iJac = inv(V(transformRefField)(cmp.refPos));
    tensor{dim} refDelta = iJac \* worldDelta; //reference velocity
    tensor{dim} npos = cmp.refPos + refDelta - refCell;
    if (K.isInside(nPos)) {
        norm = (f.transformedRefField)(x);
        grad = (f.transformedRefField)(x);
        refDelta normalize(refDelta); //reference velocity
        real eTime = K.exit(cmp, refDelta);
        if (eTime == -1) { //invalid direction, use naive scheme
            return (mesh.findPos(x.worldPos()) + worldDelta);
        } else (found = true; testp=1);
    } else {
        force = fPerp(pos)
        pos = pos + fStep(pos);
        nmp = (newE - oldE) / (rad * rad);
        pos = pos + step;

        if (found) {
            step = fStep(pos);
            pos = pos + step;
            if (force > rad) {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            } else {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            }
        } else {
            return (newE - oldE > 0.5 * (pos - oldPos))
            pos = oldPos;
            force = fStep(pos);
            nmp = nmp + step;
        }
    }
}
```

Figure 9: A complete though minimal particle system that uses guided search and is aimed at isosurfaces. We note that the figure in the paper used this program with $\varepsilon_p=0.005$, $\alpha=0.5$, and $\theta=0$. 

```c
//mesh specification:
type mesh_t = file("evalProg.json");
//space (V) specification:
type functionSpace{mesh_t} fn_v_t = file("evalProg.json");
//a(V) specification:
type femFunction{fn_v_t} func_t = file("evalProg.json");

const int dim = mesh_t::dim; //dim of mesh

input mesh_t mesh; //input mesh
input fn_v_t space = fn_v(mesh); //input space
input func_t data = func_t(space); //input u(V)
refCell{mesh_t} K = mesh.refCell; //get refCell

input int timeSteps = 12; //guided search step limit
input real timeEps = 6.000000e-01; //1 - time max
output position{mesh_t} x, y; //current mesh pos
output tensor{dim} delta; //current mesh vel

if (x.isValid) return (x);
real time = 1; //time
position{mesh_t} cmp = x; //current mesh pos
foreach (int i in timeSteps) {
    tensor{dim, dim} iJac = inv(V(transformRefField)(cmp.refPos));
    tensor{dim} refDelta = iJac \* worldDelta; //reference velocity
    tensor{dim} npos = cmp.refPos + refDelta - refCell;
    if (K.isInside(nPos)) {
        norm = (f.transformedRefField)(x);
        grad = (f.transformedRefField)(x);
        refDelta normalize(refDelta); //reference velocity
        real eTime = K.exit(cmp, refDelta);
        if (eTime == -1) { //invalid direction, use naive scheme
            return (mesh.findPos(x.worldPos()) + worldDelta);
        } else (found = true; testp=1);
    } else {
        force = fPerp(pos)
        pos = pos + fStep(pos);
        nmp = (newE - oldE) / (rad * rad);
        pos = pos + step;

        if (found) {
            step = fStep(pos);
            pos = pos + step;
            if (force > rad) {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            } else {
                return (newE - oldE > 0.5 * (pos - oldPos))
                pos = oldPos;
                force = fStep(pos);
                nmp = nmp + step;
            }
        } else {
            return (newE - oldE > 0.5 * (pos - oldPos))
            pos = oldPos;
            force = fStep(pos);
            nmp = nmp + step;
        }
    }
}
```
if (pos.isValid)
{
    // valid test replaces border control
    position cmp = nmp;
    // possibly near feature
    if (cmp.mc.meshPos(nPos)) {
        // meets feature mask
        if (fTest(cmp, nRefDelta)) {
            // passes addtl feature criterion
            // End feature functions
        }
    } else { return(normals[0]); }
}
else { return(vec3(0)); }
}

function phi(real r) {
    // r = -2.8/3;
    return(1 + r^((1.064 + r^((11.9835 + r^((-11.335 + 4.855625*r)))))
        if r < 0.0/3 else
    -0.061 + (((-0.6) + (-0.64 + (1.215 - 0.9725)*s)*s)*s)*s
        if r < 1 else 0;
    }

function phi_real (real r) { return(1 - fTest(pos)); }

real pchist = hist(1.0/2*pcp);
int iter = 0;
real rpc = 1;
int poplast = -1;
real urnd(real x) {
    if (x==0) return 0;
    1.2 - log2(x);
    real frap = 2*(floor(frapping)/2-1);
    return fmod((2^29 + 2*iter)*frap, 1);
}

return fmod((2^29 + 2*iter)*frap, 1);
}

function v3rnd(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
}

function real fStep(position mesh_t x) { return(0.01234567901*t*(4.86 + t*(-14.58 + t*(14.58 - 4.86*t)))

return fmod((2^29 + 2*iter)*frap, 1);
Figure 11: Part 2 of a particle system that uses guided search and is aimed at ridge surfaces. This section contains the core of a Diderot program:

```diderot
strand point (position(mesh_t) p0, real hh0) { //changed the type
  output position(mesh_t) pos = p0; // changed the type
  real ID = genID(p0);
  real hh = hh0;
  vec3 step = [0,0,0];
  bool found = false;
  int nfs = 0;
  real trav = 0;
  real mvmt = 1;
  real closest = rad;
  int born = 0;
  bool first = true;
  update {
    if (!posTest(pos)) { die; }
    if (travMax > 0 && trav > travMax) { die; }
    if (!found) {
      if (nfsMax > 0 && nfs > nfsMax) { die; }
      step = sfS*fStep();
      pos = pos + step;
      mvmt = lerp(step/tipd, mvmt, hist);
      if (mvmt > fsEps) {
        oldE += enr();
        vec3 npos = pos + tipd*

        npos = npos + sfs*fStep();
        trav = 0;
        nn += 1;
      } else {
        if (nn < nnmin) {
          if (birth) {
            birth = v3rnd(pos) < (nn - nnmax)/
            vec3 mno = [0,0,0];
            mno /= nn;
            mno += off;
            //to find avoid saving invalid positions:
            bool allfound = all {P.found | P in point.all};
            real percov = mean {P.closest - meancl | P in point.all };
            real covcl = sqrt(varicl) / meancl;
            real maxmav = max {P.mavm | P in point.all };
            //to find avoid saving invalid positions:
            bool allvalid = all {P.pos.isValid | P in point.all};
            real meanc1 = mean {P.closest - meancl.s | P in point.all };
            real varicl = mean { (P.closest - meancl.s)^2 | P in point.all };
            real maxmav = max {P.mavm | P in point.all };
            bool allfound = all {P.found | P in point.all};
            if (allfound &
```