Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis

Gordon Kindlmann  Charissee Chiw  Nicholas Seltzer  Lamont Samuels  John Reppy

Department of Computer Science, University of Chicago

Abstract— Many algorithms for scientific visualization and image analysis are rooted in the world of continuous scalar, vector, and tensor fields, but are programmed in low-level languages and libraries that obscure their mathematical foundations. Diderot is a parallel domain-specific language that is designed to bridge this semantic gap by providing the programmer with a high-level, mathematical programming notation that allows direct expression of mathematical concepts in code. Furthermore, Diderot provides parallel performance that takes advantage of modern multicore processors and GPUs. The high-level notation allows a concise and natural expression of the algorithms and the parallelism allows efficient execution on real-world datasets.

Index Terms—Domain specific language, portable parallel programming, scientific visualization, tensor fields.

1 INTRODUCTION

Scientific visualization and image analysis research seeks better ways to extract knowledge from scientific data. However, writing software is hard: it can be difficult to translate ideas and algorithms to working code, often due to a semantic gap between the mathematical concepts at the computational core of applications, and their actual expression in source code. This often happens with algorithms defined in terms of abstractions like fields or tensors, but implemented with library function calls or interfaces that do not reflect the basic mathematical structures. Another challenge is that visualization and analysis tools typically require parallel computing, either because new analysis methods may require more computation per iteration, or because meaningful evaluation of new methods requires their application to large real-world datasets. Datasets grow in size and complexity with continuing advances in scientific imaging modalities.

Domain-specific languages (DSLs) can address both of these challenges. By addressing a more narrowly-defined class of data types and operations than general-purpose languages, DSLs bridge the semantic gap by supporting mathematically idiomatic expression of algorithms. By targeting a specific class of algorithms, DSLs also facilitate compiling algorithms to efficient parallel execution, as well as achieving portable parallelism: mapping the same program to different parallel computing back-ends. Scientific visualization is an especially apt target for DSLs because many of its elementary abstractions are not directly supported by general-purpose languages, so the implementation of even basic methods obscures their underlying algorithmic simplicity. While many languages support vector- or matrix-valued variables, for example, we are not aware of languages that directly and idiomatically support the abstraction of a continuous scalar, vector, or tensor field, along with arithmetic and differential operations on fields. Fields, however, are fundamental to the basic definition of many core scientific visualization algorithms (e.g., ray-cast volume rendering, streamline integration, and fiber tractography).

Diderot is a portable parallel domain-specific language developed out of frustration with the lack of mathematical abstraction and the difficulty of parallel programming associated with research in scientific visualization and image analysis. Previous work [16] described an early version of Diderot, introducing its program structure, intermediate representations in compilation, computational abstraction of fields, and efficient parallel execution via pthreads. This paper presents a range of advances:

- **Portable parallelism:** With no more user intervention than setting a compiler command-line flag, Diderot can generate sequential code using SSE vector instructions, pthreads-based parallel code for shared-memory multicore systems, or OpenCL code for GPUs. Portable parallelism is increasingly useful in the context of rapidly evolving architectures for parallel computing.
- **Strand creation and communication:** The threads of execution (called strands) in Diderot can spawn new strands, strands can communicate with each other, and global reductions can be computed over the entire program. These features are important for algorithms such as particle systems that require creating dynamically new particles, and computing the interactions between particles.
- **Higher-order mathematical functions:** Fields in Diderot (scalar, vector, or tensor) can be operated on and combined as they would be in mathematics, creating new fields with expressions like \( \nabla \times V \) or \( \nabla \cdot (V \times V) \). While the previous compiler [16] had some higher-order operators, here we describe a general system for lifting tensor operations to fields and combinations of fields and tensors. Supporting idiomatic expressions of these operations simplifies implementing algorithms that depend on them.
- **Compiling to a library:** Rather than compiling to a self-contained executable, Diderot programs may be compiled to a C library with an API for setting global variables, defining the initial state of the computation, controlling program iterations, and accessing the computed output. This mechanism facilitates integrating Diderot into existing environments written in C or C++, as well as high-level languages that support C extensions.
- **Other language conveniences:** The Diderot programs shown here benefit from other new language features such as user-defined functions, dynamic sequences (e.g., for generating streamline geometry), and operations like clamp, mirror, and wrap on the convolution domains of fields.

2 BACKGROUND AND RELATED WORK

There are a number of approaches to supporting scientific visualization and biomedical-image processing applications. A common way is to build upon a domain-specific library or toolkit such as the Visualization Toolkit (VTK) [42], ParaView [2], and the Insight Toolkit (ITK) [53]. Related to libraries and toolkits are software tools that target a specific visualization method in a highly configurable way, such as the wealth of GPU-based volume rendering tools [11, 38, 34, 41].
3 LANGUAGE DESIGN

3.1 Elements of Diderot programs

This section uses Fig. 1 to introduce elements of Diderot. Appendix A gives a more detailed summary of the language. Datasets visualized or analyzed in Diderot are discretely sampled arrays of scalar, vector, or tensor data (Fig. 1(a)). The image type holds such data:

```plaintext
image(2)[0] \i = image("hand.nrrd");
```

image(2) says that the image is sampled over a 2D domain, and tensor shape specification [0] says that the samples are just scalars. 3-vectors have shape [3]; 3 x 3 matrices are {3,3}. Diderot programs do not typically operate on the image array itself, or address its individual elements. Diderot is not designed for uniformly data-parallel image computations, such as per-pixel thresholding (Fig. 1(b)).

The visualization and analysis methods in the target domain of Diderot are defined in terms of continuous scalar, vector, or tensor fields. Accordingly, Diderot programs typically start by defining a field as the convolution of discretely sampled data with a continuous reconstruction kernel. For example,

```plaintext
field(#1(2)[0] \f = ctmr @ image("hand.nrrd");
```

defines a $C^1$ continuous ("$\#1$") scalar ("$\{0\}$") field $\mathbf{F}$ over a two-dimensional ("$\{2\}$") world-space, by convolving ("@") the image data in hand.nrrd with the Catmull-Rom cubic spline ctmr. Fig. 1(c) shows a dense sampling of this field.

Diderot programs are decomposed in terms of strands, either autonomous or interacting, which can move throughout fields. Fig. 1(d) shows a dense sampling of an isocontour of $\mathbf{F}$ found via Newton-Raphson iteration. Isosurface sampling in Fig. 1(d) is computed by:

```plaintext
1  strand isofind (vec2 pos0) {
2    output vec2 x = pos0;
3    int steps = 0;
4    update {
5      // Stop after too many steps or leaving field
6      if (steps > stepsMax || !inside(x, F))
7        die;
8      // one Newton-Raphson iteration
9      vec2 delta = -normalize(\nF(x) + \nF(x)/\nF(x));
10     x += delta;
11     if (\n|delta| < epsilon)
12       stabilize;
13     steps += 1;
14   }
15 }
```

The heart of a Diderot program is the “update” function, which expresses one iteration of the algorithm. Strands can die to terminate without saving any output, or stabilize to save the most recently
Input:

```diderot
Fig. 2. Complete Diderot program for particle-based isocontour sampler (Fig. 1(e)), a simple 2-D version of the method of Meyer et al. [33], without curvature-dependent density. At a high level, the program starts with (lines 1-6) the declaration of global inputs, to be set either on the command-line when compiling or at a stand-alone executable, or via API when compiling to a library. After creating (line 8) the scalar field in which the isocontour \( F(x) = 0 \) will be sampled by interacting particles (according to pairwise energy \( \phi \) and force \( \phi' \)), the strand definition includes local strand state (lines 15-19) and the `update` method, which proceeds in two stages.

1. **Input preparation**: The program begins by defining global inputs such as `radius`, `positionLast`, and `force`. These inputs are used to set up the initial conditions for the simulation.

2. **Field initialization**: The scalar field is created using the `field2[]` function, which takes a function `F` and an array of image data. This field represents the isocontour `F(x) = 0`.

3. **Iteration loop**: The main loop of the program iterates over particles, updating their positions (`pos`) and velocities (`force`). The loop continues until the desired number of iterations is reached.

4. **Particle interaction**: Particles interact with each other and with the scalar field to compute the force acting on each particle. The `foreach` loop (line 31) iterates over each particle, and the `normalize` function is used to ensure that the force vectors are normalized.

5. **Energy calculation**: The energy of the system is calculated using the `phi` function, which represents the interaction potential between particles. The derivative of this function, `phi'`, is also computed.

6. **Update step**: After calculating the force and energy, the particle positions are updated using a gradient descent method. If the force is zero, the particle is considered stationary, and the program moves to the next iteration.

7. **Convergence check**: The program checks for convergence by comparing the `motion` (the change in position) to a threshold value `epsilon`. If the change is below the threshold, the iteration is considered complete.

8. **Particle removal and re-insertion**: If a particle has no neighbors, it is removed from the list and re-inserted nearby to fill in any gaps in the isocontour.

9. **Output generation**: The final positions of the particles are stored in a list, and the program outputs these positions as the result of the computation.

The ease of Diderot programming is fostered by notational and structural aspects of the language. The mathematical notation we write on whiteboards should work in programs, so Diderot supports tensor fields as computational values, and uses Unicode to idiomatically express mathematical operations. At a structural level, computation in Diderot is organized into *strands* according to the structure of the algorithm *output*. The output may be an image (in the case of volume rendering), or a set of polylines (for streamlines), or simply a point set (for particle systems). For these examples, the Diderot program states the work required of each strand to produce one rendered pixel, one streamline, or one particle, respectively. This increases the flexibility of Diderot relative to languages that structure parallelism around the input data domain (pixels or voxels).

The second goal is to efficiently execute Diderot programs by using modern parallel hardware, such as GPUs. Diderot’s execution model attempts to achieve expressiveness while avoiding features that would be difficult to map onto parallel hardware, or that would force programmers to worry about parallel programming challenges like data races, synchronization, and locality. We have therefore based the semantics of Diderot on a simple deterministic parallel programming model, described in Sec. 3.3. This model fits the iterative structure of typical scientific visualization algorithms, while allowing some flexibility in how the model is mapped to different hardware targets.

### 3.3 Computing with tensors

Diderot strives to mimic the “direct” or coordinate-free notation for vector and tensor expressions commonly used in continuum mechanics [27]. This motivates using \( u \otimes v \) for the tensor product, instead of something like “\( u \cdot v \)”, which depends on recognizing \( u \) and \( v \) as N-by-1 column vectors, and which does not generalize to higher-order tensors. Expressing computations in a coordinate-free way tends to clarify mathematical intent. The direct notation encouraged by Diderot also fosters dimensionally general code. For example, to turn Fig. 2 into a program to sample isosurfaces in *three* dimensions, the only changes required are replacing `vec2` and `field2[]` with `vec3` and `field1[3][]`, respectively, and adding a dimension to the strand initialization (line 72).

### 3.4 Reconstructing and computing with fields

Fields are created by convolving (noted with `@`) data on regularly sampled grids with continuous reconstruction kernels. Diderot currently knows a variety of kernels that differ in their continuity and

---

1Unicode character entry is a possible concern, but the range of practical solutions is expanding: [http://en.wikipedia.org/wiki/Unicode](http://en.wikipedia.org/wiki/Unicode)
support (how many samples along each axis are needed in convolution). Besides the ctmr Catmull-Rom C^1 cubic spline (4-sample support) used in Fig. 1, other possible kernels include C^0 linear interpolation tent (2 samples), C^3 cubic B-spline bspln3 (4 samples), and a C^4 piecewise hexic kernel ehxlic (6 samples) that accurately reconstructs cubic polynomials (a 4th-degree error filter or 4EF in the terminology of [36]).

Reconstruction of fields in Diderot is by separable convolution; the same kernel is convolved with each image data axis. For example, the convolution \( F = V \otimes h \) of image data \( V[i, j] \) with \( s \)-sample support kernel \( h \) is evaluated at image-space location \((x_1, x_2)\) with

\[
F(x_1, x_2) = \sum_{i,j} V[i,j]h(x_1-i)h(x_2-j);
\]

(1)

\[
n_s = |x| \Rightarrow \alpha_i = x_i - n_s \in [0, 1) \Rightarrow
\]

(2)

\[
F(x_1,x_2) = \sum_{i,j=-1/2}^{s/2} V[n_1+i,n_2+j]h(\alpha_1-i)h(\alpha_2-j).
\]

(3)

The statement (1) of separable convolution typical in image processing and computer graphics [35] shows how the continuity order of field \( F \) is the same as that of reconstruction kernel \( h \) and that partial derivates \( \partial F / \partial x_i \) are reconstructed by convolving with kernel derivatives \( h' \) on one axis. Diderot uses (3) to limit the convolution to the kernel support. A field \( V \) in Diderot is defined over world-space and evaluated with \( F(p) \) at world-space position vector \( p \). The field implementation includes the homogeneous coordinate matrix \( M \), transforming index-space location \( \bar{x} \) to world-space \( \bar{p} \), to represent the image orientation learned from the image data file. Field evaluation \( F(p) \) first computes \( \bar{x} = M^{-1}\bar{p} \) (the inverse \( M^{-1} \) is pre-computed once) before applying (3). Further details of the convolution implementation are in [16].

Diderot supports differential and arithmetic operators on fields in a mathematically idiomatic way. Evaluation of the field derivative \( \nabla F(x) \) correctly returns the gradient in world space according to the covariant transform of the index-space partial derivatives \( \partial F / \partial x_i \). The derivative of a field \( C \) of order-\( n \) tensors (\( n > 0 \)) is field \( \nabla \otimes C \) of order \((n+1)\) tensors. A compile-time type error results from differentiating a field beyond its continuity order, like the (second derivative) Hessian \( \nabla \otimes \nabla \) of a \( C^2 \) scalar field \( \text{field}(3)[\atilde]F \).

While thinking mathematically about visualization algorithms on fields, it is natural for programmers to consider expressions like \( f = \text{u} \times \text{v} \) encoding the understanding that \( u \), \( v \), and \( f \) could stand for constant values, or fields, or some mix of both. Diderot syntax supports this convention. As defined in Sec. A.10, arithmetic operators like addition, subtraction, scalar multiplication \( \times \), and inner product \( \bullet \) can work on a mix of tensors and tensor fields. Two fields arising from image data with different orientations and resolutions may be added without restriction. Furthermore, as described in Sec. 5.2, differential operators like gradient \( \nabla \), Jacobian \( \nabla \otimes \), curl \( \nabla \times \), and divergence \( \nabla \bullet \) may be applied to such mathematically constructed fields, and the Diderot compiler determines how to compute their evaluation.

3.5 Bulk-Synchronous Parallel Strand Execution

The bulk-synchronous parallel (BSP) programming model of Valient [49, 44] provides the basis of Diderot’s execution semantics, illustrated in Fig. 3. All active strands execute in parallel execution steps called super-steps, each with two phases: a strand update phase and an optional global computation phase. First, each active strand executes its update method once, to update its local state, including output variables. Strands do not directly communicate, but they may learn the state of nearby strands according to their position in world space, as used in the particle system code in Fig. 2. A strand makes a spatial query to get an iterable sequence of all strands within some world-space radius, providing (read-only) access to their state at the beginning of the super-step. This requires maintaining two copies of strand state, but we can optimize away the extra storage in many cases. Diderot is not intended for applications in which all pairs of strands need to directly communicate, and of the possible mechanisms for limiting strand communication we choose spatial proximity. The assumption is that each strand is responsible for analyzing or visualizing some local region of the data (as with particle systems), so any communication required between strands can likely be expressed in terms of region (or strand) neighborhoods.

Second, the global phase computes reductions over strands (such as finding average or extremal values of strand variables) to update global variables, or to terminate program execution by stabilizing all strands. At the start of the next super-step, strands created by new start executing; and all strands can read (but not write) values of globals set in the preceding global phase. Each strand is idle after it finishes executing its update method until the end of the strand update phase. Stable strands are idle for the entirety of their update. Dead strands are also idle during update, but do not produce any output. The program executes until all strands are either stable or dead. The BSP model is deterministic because the order of strand scheduling does not affect the computation. In practice, non-determinism may arise from the non-associativity of floating-point arithmetic in global reductions, but for debugging purposes this can be avoided at some performance cost.

3.6 Programs as libraries

One drawback of DSLs is their lack of infrastructure for I/O, graphics, networking, or other services required of complete applications. Such functionality requires significant implementation effort and may dilute the domain-specific focus of the language. One solution is to embed the DSL in a general-purpose host language, so the embedded DSL can use all the features of the host language. Delite takes this approach, for example [15, 10]. We did not embed Diderot because we did not want to commit to a single host language, and be constrained by its syntax and type system. Supporting idiomatic expression of mathematical operators required Unicode, and field properties like continuity order and tensor shape are not readily encoded in other languages.

To address the infrastructure problem, we compile Diderot programs to libraries with a C-language API. This allows programmers to develop applications in any language that can call C code. The API includes functions for setting inputs, reading outputs, and controlling program execution. A Diderot program can either be run to completion or for some number of super-steps. Snapshots of program state can be accessed to create animations of program execution. The compiler can additionally generate stand-alone executables (with command-line options to set inputs), which link against the C library created by the compiler. Appendix B demonstrates the API generated from a program for isocountour sampling (Fig. 1d).
tools, by exploiting Diderot’s ability to operate on continuous fields as first-class mathematical objects. The quantities volume rendered in these examples (scalar field Canny edges, flow field vortex structure indicators, and tensor invariants) are all defined with a few lines of Diderot code, and the subsequent volume rendering code remains the same. Even though the code works from an underlying scalar-, vector-, or tensor-valued volume dataset, the rendering is always of a scalar field analytically derived from the data (not pre-computed on a grid) and numerically evaluated at each sample along each ray. The Diderot language permits a clean separation between the code that implements the core visualization algorithm, and the code that defines the quantity of interest to visualize.

```plaintext
1 input real isoval;
2 input real thick;
3 input vec3 CamEye; input vec3 camAcT; input vec3 CamUp;
4 input real camFOV; input int isovalI;
5 input real refStep; input real rayStep;
6 input vec3 LightVsp; // light direction in view-space
7 input real phongKa; input real phongKd;
8
9 field4([],[]) V = bspln5 @ image("ctscan-prefilt.nrrd");
10 field4([],[]) F = V - isoval; // isosurface is |x|F(x)|=0
11 function real mask(vec3 x) = 1.0;
12 function vec3 color(vec3 x) = cmap(F(x));
13 function real alpha(real v, real g) = u = clamp(0, 1, 1.3*(1 - |v|/(g*thick)));
14 real camDist = |camAt - camEye|;
15 vec3 camN = normalize(camN = camN × camUp); // away
16 vec3 camO = normalize(camO = camN × camUp); // right
17 vec3 camU = normalize(camU = camN × camN); // up
18 vec3 camFar = tan(camFOV*π/360) × camDist;
19 real camNear = clamp((camO × camN)/norm(camO × camN), 0.1, 1.1) × camDist;
20 vec3 light = transpose((camU × camN) × normalize(lightVsp));
21
22 isovale = raycast (int u, int v) {
23 real rayU = lerp(-camN × camN, camN, -0.5, u, isovalI-0.5);
24 real rayV = lerp(-camV × camN, camV, -0.5, v, isovalI-0.5);
25 real rayN = rayV × camN + rayU × camN / camDist;
26 real transp = 1;
27 vec3 rgb = [0, 0, 0];
28 output vec4 rgb4 = [rgb[0], rgb[1], rgb[2], transp];
29 update {
30 vec3 x = camEye + rayN × rayV;
31 if (inside(x, u, v)) {
32 real val = F(x);
33 vec3 grad = -∇V(x);
34 real alpha1 = alpha(val, |grad|) × mask(x);
35 if (a > 0) {
36 a = 1 - pow(-rayN × rayV / rayStep, 2); // undo pre-multiplied alpha
37 rgb += transp × a × (phongKa + phongKd × shade) × color(x);
38 transp = 1 - a;
39 }
40 }
41 if (transp < 0.01) { // early ray termination
42 transp = 0.1;
43 stabilize;
44 }
45 if (rayN > camFar) stabilize;
46 rayN = rayN + rayStep;
47 }
48 stabilize;
49 real a = 1 - transp; // undo pre-multiplied alpha
50 if (a > 0) rgb4 = [rgb[0]/a, rgb[1]/a, rgb[2]/a, a];
51 initial isovale = rgb4;
52
53 field2([],[]) V = -V(|V|) • V/|V|;
54 function real mask(vec3 x) = 1.0 if (|V(x)| > gmin) else 0.0;
```

Fig. 4. Complete Diderot program for volume rendering an approximate isosurface at isovalue `isoval` with thickness `thick` in scalar field `F`.

Fig. 4 shows the complete listing of a Diderot program for ray-cast volume rendering. While the code is mostly self-explanatory, lines 8 to 14 merit further explanation, since they will be changed in subsequent examples. The CT scan volume dataset `ctscan.nrrd` used on line 8, and all subsequent volume datasets used in these volume rendering examples, are pre-filtered so that convolution with the 5-order B-spline `bspln5` (piece-wise quintic, $C^4$ continuous, 6-sample support) interpolates [48]. The scalar field in which to render the zero isocontour is (line 9) $F=V-\text{isoval}$. The per-sample color and opacity in the ray-casting are defined in terms of the cmap colormap (1-D field of 3-vectors, line 11) and the alpha bivariate opacity function (13) based on Levoy [30].

Fig. 5 shows a CT scan of a *Cebus apella* (capuchin) head, for which an image analysis goal involves recovering the bone surface. Diderot can visualize the bone surface by volume rendering. The left side of Fig. 5 shows isosurfaces rendered with the code in Fig. 4, at three isovales that span the range of CT values for bone. At too low an isovale (V=1300) some of the soft tissue is visible, but at a higher isovale (V=2150) suitable for most of the skull, holes incorrectly appear at the frontal sinuses (above and between the eye sockets). The tooth surface is cleanest at yet another isovale (V=3000). The colormap is approximately isoluminant.

A classical principle of edge detection proposed by Canny is that edge points are where the image gradient magnitude is maximized with respect to motion along the (normalized) gradient direction [14]. Algorithmically, Canny edge detection also involves finding optimal smoothing, and hysteresis thresholding of edge components based on gradient magnitude. These are outside the scope of Diderot, but we can still capture Canny’s principle of edge localization. In scalar volume data field $V(x)$, we seek locations where $|\nabla V(x)|$ is maximized with respect to motion along $\nabla V(x)/|\nabla V(x)|$, which means that $\nabla V(x)$ (the gradient of quantity being maximized) is orthogonal to $\nabla V(x)/|\nabla V(x)|$. In Diderot, we write:

```plaintext
9 field2([],[]) F = -V(|V|) • V/|V|;
10 function real mask(vec3 x) = 1.0 if (|V(x)| > gmin) else 0.0;
```
in place of lines 9 and 10 in Fig. 4. Strong edges (as opposed to minima of gradient magnitude) are selected by user-specified threshold gmin. The right of Fig. 5 shows the results, with no other changes to the code of Fig. 4. Note that the Canny edges are shaded correctly, based on $V$. The isocontour shading also used $V$ (line 41), but now $V$ is some expression involving third derivatives of the volume data $V$, which the Diderot compiler generates automatically, based on its understanding of vector calculus. The smoothness apparent in the bone surface is thanks to the $C^4$ continuous reconstruction of the data field `field4([],[]) V` (line 8). The Canny function `field2([],[]) F` is $C^2$ continuous (based the second derivative of $V$), and its gradient $V$ will be $C^1$ continuous. With no prior segmentation or parameter tuning, Fig. 5 correctly shows the skull surface over the frontal sinuses, as well as the very thin bones of the orbital walls (within the eye socket), neither of which are visible with any isovale reliably higher than all the soft tissue. Diderot has thus greatly simplified the successful visual exploration (by volume rendering) of a first-principles approach (Canny edges) to bone surface extraction in real-world three-dimensional image data.

Flow visualization is another domain of scientific visualization that involves mathematically sophisticated consideration of vector fields and their derivatives. For example, one statement of vortex structure

![Fig. 5. Volume rendering of isocontours (left) and Canny edges (right) from a CT scan of *Cebus apella* (capuchin) head.](image-url)
identifies them with locations where the direction of flow \( \frac{\nabla}{|\nabla|} \) is aligned with that of its curl \( \nabla \times |\nabla| \) [20, 37]. This is equivalent to saying that the normalized helicity \( \frac{\nabla \times |\nabla|}{|\nabla|} \) is at an extremum (near +1 or -1). We show here how volume rendering in Diderot can directly visualize the mathematical elements of this type of flow field analysis.

The renderings in Fig. 6 use a single time-step from a Navier-Stokes simulation of flow (from left to right) past a square rod, creating a train of vortices [13, 51]. The Fig. 4 volume rendering code is re-used, but with \( \nabla \) defining a 3-D vector field:

```diderot
8 field(3)[3] V = bspline @ image("flow.nrrd");
```

and with a diverging colormap of the Jacobian discriminant (in lieu of lines 11 and 12):

```diderot
field(3)[3] J = VxV;
field(3)[3] A = -trace(J);
field(3)[3] B = (trace(J) * trace(J) - trace(J * J))/2;
field(3)[3] C = -det(J);
field(3)[3] Q = (A*A/3.0 - B)/3.0;
field(3)[3] R = (-2.0*A*A*A/27.0 + A*B/3.0 - C)/2.0;
field(3)[3] D = R*R - Q*Q*Q; // the discriminant
field(3)[3] dmap = clamp(tent @ image("diverg.nrrd"));
function vec3 color(vec3 x) = dmap(D(x));
```

After forming a field of the Jacobian \( \nabla \times \nabla \), fields \( A \), \( B \), and \( C \) are the coefficients of the characteristic polynomial \( p(\lambda) = \lambda^3 + 3A\lambda^2 + 3B\lambda + C \), and the discriminant field \( D \) is the pre-cursor to analytically solving \( p(\lambda) = 0 \) for \( \lambda \). There are two complex-conjugate eigenvalues (indicating rotational flow) when the discriminant \( D(x) \) is negative [37].

The only substantial difference in how the various renderings in Fig. 6 were computed was in the statement of the derived scalar field \( \Phi \) being rendered. Fig. 6(a) shows simple isosurfaces of flow magnitude, created by using

9 \text{field}(3)[3] F = |V|;

Fig. 6(b) visualizes isosurfaces (at \( \pm 0.99 \)) of a quantity related to the extremal lines of the field: places where the vector magnitude is extremal with respect to motion perpendicular to the vector, or, where \( \nabla|\nabla| \) is aligned with \( |\nabla| \) [46, 37]:

9 \text{field}(3)[3] F = \nabla|\nabla| \bullet (\nabla |\nabla| / (\nabla |\nabla|));

Fig. 6(c) visualizes isosurfaces (at \( \pm 0.99 \)) of normalized helicity, using

9 \text{field}(3)[3] F = \nabla|\nabla| \bullet (\nabla \times |\nabla| / (\nabla \times |\nabla|));

Note that in all cases, the Diderot compiler is determining how to analytically differentiate field \( F \) so that it can be shaded correctly. These renderings use two-sided lighting:

41 \text{real shade} = \text{normalize}(\text{grad}) \bullet \text{light};

It should be emphasized that although a scalar field is being volume rendered to make these images, there is no pre-computed scalar dataset: the scalar field is defined symbolically in terms of a vector field (reconstructed by convolution), and the Diderot compiler generates the necessary instructions to probe the scalar field at each ray sample position during rendering.

Two examples of volume rendering tensor fields further demonstrate the simplicity of creating mathematically and computationally sophisticated visualization tools with few lines of Diderot code. Research in tensor field topology often investigates lines of degeneracy: locations in a tensor field where two of the tensor eigenvalues are equal, or equivalently, places where the tensor mode is \( \pm 1 \) [19, 47].
The mode of tensor $D$ is defined as

$$\text{mode}(D) = 3\sqrt{\det(D/D^2)}$$ (4)

$$D = D - \text{tr}(D)/3$$ (5)

$$\bar{D} = \sqrt{\text{tr}(D/D^2)}.$$ (6)

This can be directly expressed in Diderot, starting with a stress tensor dataset (simulating a double point load), then defining fields of tensor deviatoric $E$ and mode $F$:

- \text{field}(3,3) V = \text{bspln5} @ \text{image}(\text{"stress.nrrd"});
- \text{field}(3,3) E = V - \text{trace}(V) \cdot \text{identity}[3]/3;
- \text{field}(3,3) F = 3 \cdot \sqrt{6} \cdot \text{det}(E/E^2);$

Fig. 7(a) shows $F$ rendered with the same code as in Fig. 6, with yellow for mode $= +1$ and blue for mode $= -1$. The rendering shows the lines of degeneracy (the ridge and valley lines of tensor mode) [47].

Finally, Fig. 7(b) shows a volume rendering of half of a human brain diffusion tensor scan, showing an isosurface of fractional anisotropy (FA), which quantifies the amount of directional organization in the white matter as captured by the single tensor model of diffusion. Though FA is often defined in terms of eigenvalues, its original definition [5] involved just the tensor $D$ and its deviatoric $D$ (5)

$$FA = \sqrt{2/3} D$$ (7)

Diderot permits directly translating this definition into working code:

\begin{verbatim}
fld4(3,3,3) V = bspln5 @ image("stress.nrrd");
field(3,3) E = V - trace(V)*identity[3]/3;
field(3,3) F = 3*sqrt(6)*det(E/E);
\end{verbatim}

The RGB coloring is provided by the traditional map of the principal eigenvector, modulated by linear anisotropy [52]. In Diderot:

```
function vec3 color(vec3 x) {
    real[3] ev = evals(V(x));
    vec3 dir = evvecs(V(x))[*0];
    real CL = (ev[0] - ev[1])/ev[0];
    return [dir[0]*, dir[1]*, dir[2]*]*CL;
}
```

Note also that in both tensor invariant (mode or FA) volume renderings, the Diderot compiler did the “heavy lifting” of analytically deriving and numerically computing the spatial derivatives of the tensor invariants, expanded in terms of the spatial derivatives of the individual tensor components. Manually deriving and implementing these expressions is tedious and error-prone [47].

4.2 LIC and Streamlines

Fig. 8 lists a Diderot program for computing a line-integral convolution (LIC) visualization of a two-dimensional vector field, seen as the background image of Fig. 10. The program is a straightforward implementation of the original LIC method [12], in which samples of a noise texture (field $h$) are averaged along a streamline through normalized vector field $nV$ (line 6). The integration continues for $\text{stepNum}$ steps or until the streamline leaves the domain. Field evaluations falling outside the domain are gracefully handled by the border functions $\text{clamp}$ (line 5) and $\text{wrap}$ (line 6). Upon stabilization, the output color value is computed by modulating the LIC contrast by the velocity at the start point $x_0$ (line 23) and colormapping by vorticity $V \cdot V(x_0)$ (line 24). Over-all image contrast is clamped to encompass roughly two standard deviations (line 25) in the convolution result, where $\text{stdv}$ is computed (line 5) as the expected standard deviation of the average of $\text{stepNum}$ samples in the unit standard deviation noise field. Two strands are used for each output pixel; one upstream (line 32 $\text{si}=0$ and line 9 $h=-h$) and one downstream ($\text{si}=1$ and $h=h$).

To emphasize (along with the particle system in Fig. 1(e)) that Diderot programs can generate geometry as well as values on grids, the program in Fig. 9 computes the geometry of the streamlines overlayed on the LIC result in Fig. 10. For each seedpoint $x_0$ in text file seeds.txt (line 1), $\text{sline}(x_0)$ outputs a sequence of vec2 streamline vertex positions, starting with $x_0$ (line 9). As they are computed by midpoint method integration (line 12), points $x$ along the streamline are appended to the output path (line 13). A small arrowhead, proportional to $V(x)$ (line 18), is added in the $\text{stabilize}$ method to the downstream end of the streamline polylines; this is converted upon rendering (not shown) to a filled triangle. In the program output, all polylines vertices are concatenated into a single array, and a second array stores the start indices for each streamline. The polylines were post-processed to produce the vector graphics seen in Fig. 10.

5 IMPLEMENTATION

5.1 Strand communication and global reduction

Streamlines may communicate with neighboring strands according to their proximity in world space. We accelerate the spatial queries with a k-d tree [24], a binary tree that at each internal node splits world space along one axis. The split axis alternates for two dimensions) or cycles (three dimensions) through the axes as one descends the tree. Strands are assigned to nodes in the tree by comparing their coordinate on the splitting axis to the splitting value.

Spatial queries pose an implementation challenge for parallel targets. To account for strand motion, and the creation of new strands, the tree must be maintained at the end of each super-step, prior to executing the next strand updates. Maintaining the spatial data structure must be parallelized to avoid sequential bottlenecks. We use a parallel version of the median-of-medians algorithm [7] to select the splitting value for the world-space coordinate at each level of the tree. The GPU presents additional LICs for the dynamic memory management needed to support strand motion and creation. OpenCL requires that GPU memory be preallocated on the host side before running the GPU computation, but an execution step might exceed the allocation. For example, if the host side preallocated memory for 30 strands, used to store the states of 20 active or stable strands, then the current execution step cannot successfully create more than then 10 new strands. Fortunately, Diderot’s execution model preserves the starting state of strands in a given execution step, so we can abort the step if we run out of memory and restart it with a larger preallocation.
Strands may indirectly share information via global reductions (the second part of the super-step), which can modify the global variables that strands may read in their subsequent update. We accelerate the reductions by regrouping them to avoid needless repeating work. For example, if a mean and product reduction can be performed together then the Diderot compiler will group them into the same execution phase rather than executing them individually. The regrouping is straightforward to implement in the sequential version; the main challenge is fusing reduction phases to reduce overhead. For the parallel version, we integrate the reductions into the barrier synchronization convention, a concise notation for tensor calculus [23]. Others fits of domain-specific optimizations without having to encode a lot of domain-specific knowledge in the compiler.

5.2 EIN Intermediate Representation

Relative to [16], Diderot has a new intermediate representation (IR), called EIN. EIN is inspired by Einstein index notation or the summation convention, a concise notation for tensor calculus [23]. Others have previously extended Einstein notation, based on close study of its ambiguities and limitations [26, 1, 4, 18, 43, 45, 21]. Part of the ambiguity is related to implicit summation. EIN uses explicit summation, and adds representation of operations such as convolution, image indexing, kernel differentiation, and trigonometry.

The EIN IR is embedded Diderot’s static single assignment (SSA) representation, with EIN assignment nodes of the form

\[ t = \lambda \text{params} \langle e \rangle \alpha (\text{args}) \]

where \( t \) is the variable being assigned, \( \lambda \text{params} \langle e \rangle \alpha \) is an EIN operator with formal parameters \( \text{params} \) and body \( \langle e \rangle \alpha \), and \( \text{args} \) are the arguments to the EIN operator. Once a surface language operation is mapped to an EIN operator, the compiler can handle the computations generically, by systematically applying EIN operators to one another, normalizing, and optimizing. The compiler can then break the EIN operator apart into simple and direct scalar and vector operations to generate code.

A family of operations (like tensor plus tensor, field plus tensor, field plus field) that previously [16] required several type-specific operations and case analysis can be captured with a single EIN operator. All tensor operators are lifted to tensor fields as consequence of their expression in EIN. The previous compiler generated code for a fixed set of tensor types, and a fixed set of operations between them, which impeded adding new operators to Diderot. EIN represents tensors and tensor fields of arbitrary shape. The new compiler simplifies complicated and large EIN operators into operations on scalars and vectors. It needs only to generate code for scalar and vector operations, avoiding the explosion of shape-specific operators in the compiler.

The compiler uses a rewrite system to optimize and lower the EIN representation. The Diderot programmer can then define fields with expressions like \( \nabla (F \circ G) \) without worrying about how to expand and simplify this in terms of derivatives of \( F \) and \( G \), and their individual scalar components. Rewrite rules include derivative identities such as

\[ \nabla (F + G) \Rightarrow (\nabla F + \nabla G) \]

\[ \nabla (f \circ g) \Rightarrow g \nabla f + f \nabla g \]

the quotient rule, and the chain rule. EIN also includes the permutation tensor \( \delta_{ik} \) and the Kronecker delta \( \delta_{ij} \). The compiler understands that \( \delta_{ik} \delta_{jk} \Rightarrow 0 \), where \( \delta_{jk} \) indicates partial derivatives along axes \( j \) and \( k \), which produces well-known identities like \( \nabla \times \nabla \phi \Rightarrow 0 \) and \( \nabla \cdot (\nabla \times F) \Rightarrow 0 \). The compiler also understands that \( \delta_{ijk} \delta_{ilm} \Rightarrow \delta_{lj} \delta_{km} - \delta_{jm} \delta_{li} \) and \( \delta_{ij} T_j \Rightarrow T_i \), which produces identities such as \( (a \times b) \times c \Rightarrow b(a \cdot c) - a(b \cdot c) \). Being able to automatically find identities as part of optimization provides the benefits of domain-specific optimizations without having to encode a lot of domain-specific knowledge in the compiler.

6 PERFORMANCE EVALUATION

While space does not permit an in-depth performance analysis, we revisit previously published benchmarks [16] to demonstrate significant performance improvements provided by the new compiler. These
results address our second design goal (Sec. 3.2) of combining good parallel performance with our high-level programming model.

Our test machine is a dual Intel Xeon E5-2687W system (16 cores) running Ubuntu 12.04 Linux. The OpenCL measurements were taken on the same system using a NVIDIA Tesla K20c with NVIDIA’s CUDA 6.0 driver. All code was compiled with optimization level -03. For each benchmark, we report the average wall-clock time for the computational part from 10 runs on a lightly-loaded machine.

The benchmarks presented here are from our earlier work, and were originally chosen to represent typical workloads:

- **textbf:** a simple volume renderer with Phong shading, like Fig. 4.
- **illus-vr:** A more complex volume renderer with curvature-based illustrative rendering [29], including tensor calculations that are awkward to express in other languages.
- **lic2d:** Line integral convolution visualization of a synthetic 2D vector field [12], like in Fig. 8 but simpler.
- **ridge3d:** An initial uniform distribution of points within a portion of CT scan of a lung is moved iteratively towards the centers of blood vessels, using Newton optimization to compute ridge lines [22], requiring eigenvalues and eigenvectors of the Hessian.

The most striking improvement of our current system over our previously reported work is in the performance scaling on SMP hardware. Fig. 11 shows the speedup curves for our benchmarks using both the current version of Diderot (solid lines) and the previous version (dashed lines). We now get excellent speedup for all four benchmarks at 16 cores, whereas the previous version scales poorly and does significantly worse on three benchmarks. Furthermore, the baseline performance of the previous version is worse, often significantly, so we are seeing better scaling even when compared to a better baseline.

Tbl. 1 presents selected numeric results from our experiments. For each benchmark, we report several sets of numbers: the execution time for the Teem version (hand-coded C), sequential and parallel (1, 6, 12, and 16-core) execution times for the PPLD’12 version of the system, sequential and parallel (1, 6, 12, and 16-core) execution times for the current version of the system, and execution times for the OpenCL version of the program. The sequential performance of our compiler has improved — 15% faster for three of the four benchmarks.

The performance of the OpenCL target is benchmark dependent. For a program like illust-vr, which has significant arithmetic intensity, the GPU is comparable to 9-core performance. On the other hand, it performs much worse on lic2d, which is more memory-bound. Appendix C compares Diderot with hand-written OpenCL code for two kinds of volume rendering. These limited results suggest that non-trivial programs are more easily expressed in Diderot than in OpenCL, with a performance penalty that is acceptable when the coding skill and time of researchers is the actual bottleneck. Nonetheless, there is room for significant improvement in our OpenCL implementation.

7 DISCUSSION AND ONGOING WORK

This paper presents a significant step towards creating a powerful and portable parallel DSL for scientific visualization and image analysis. In particular, supporting fields as a fundamental abstraction in Diderot frees programmers from worrying about the computational details of convolving on the underlying sampling grid and the mathematical details of differentiating arithmetically constructed fields.

The utility of Diderot would be enhanced by further work on its mathematical basis. We hope to expand the kinds of fields it can handle. Sec. 4.1, for example, used the same volume rendering code to visualize a variety of fields (all based on convolution on regular grids). The same rendering code could in principle be re-used to visualize fields arising from higher-order basis functions on unstructured meshes, point clouds with radial basis function, or analytic closed-form expressions. More ambitious would be a way of defining the update method not with imperative program statements, but with a declarative statement of mathematical intent. The volume rendering code of Fig. 4 would be simplified by stating its goal as solving a volume rendering integral, if the sampling and numerical integration schemes could be automatically generated by the compiler. Diderot cannot currently warn the user, for example, when too large a ray sampling distance will cause undersampling artifacts.

Diderot will also benefit from work in less mathematical areas. Most important are improvements to our implementation on GPUs. These include developing a better scheduler for strands on the GPU; our current scheduler does well with irregular workloads, but its baseline performance is not very good. We are also looking to implement a virtual memory strategy akin to that already developed for GPU-based rendering [25]; such a scheme will allow us to scale to data sets that are larger than the available memory on GPUs. Further work on code generation may lead to other GPU performance improvements. Our compiler currently generates OpenCL code, but generating CUDA code should allow better performance on NVIDIA hardware. Targeting OpenCL or CUDA, instead of assembly code or low-level GPU code (e.g., SPIR or PTX), greatly simplifies our Diderot compiler implementation, while benefiting from the optimizing compilers specialized for the target hardware. Finally, automatically generating GUIs for the input and output variables of a Diderot program would accelerate the exploration of the parameter space of different algorithms.

The source code for the Diderot compiler is available and a release is upcoming, which will merge features currently split across different branches. Scripts for regenerating representative images from freely available datasets will be made available from our web page.

ACKNOWLEDGMENTS

We gratefully acknowledge the anonymous reviewers for their constructive comments. We also thank the providers of data seen in the figures. Fig. 1: University of Utah SCI group, NIH NIGMS grant P41GM103545. Fig. 5: Callum Ross, University of Chicago. Fig. 6: Resampling by Tino Weinkauf of Navier-Stokes simulation by S. Carrafiello. M.-V. Salvetti, M. Buffoni, and A. Iollo [28]. Fig. 7(a): Xavier Tricoche, Purdue University. Fig. 7(b): Centre for Functional MRI of the Brain, John Radcliffe Hospital, Oxford University. Fig. 10: Wolfgang Kollmann, UC Davis. Portions of this research were supported by National Science Foundation award CCF-1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

REFERENCES


http://diderot-language.cs.uchicago.edu
A THE DIDEROT LANGUAGE

This appendix provides a concise, but complete tour of the Diderot language.

A.1 Types

Diderot is a strongly-typed language with a novel, dependent-type system that statically tracks important mathematical properties of the program. The following table summarizes the types of the language:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>booleans</td>
</tr>
<tr>
<td>int</td>
<td>integers</td>
</tr>
<tr>
<td>string</td>
<td>strings</td>
</tr>
<tr>
<td>tensor</td>
<td>tensor with shape ( \sigma )</td>
</tr>
<tr>
<td>image</td>
<td>dimension-d image ( (1 \leq d \leq 3) )</td>
</tr>
<tr>
<td>kernel( k )</td>
<td>reconstruction kernel of continuity ( k (0 \leq k) )</td>
</tr>
<tr>
<td>field( k )</td>
<td>tensor field with continuity ( k ), dimension ( d ), and shape ( \sigma )</td>
</tr>
</tbody>
</table>

\( \sigma := d_1, \ldots, d_n \) tensor shape \((0 \leq d)\)

Diderot also defines synonyms for common tensor shapes, such as real for tensor[1] and vec3 for tensor[3].

A.2 Top-level definitions

A Diderot program consists of a sequence of top-level definitions. At a minimum, the top-level definitions include a strand definition and a global initially block, but they may also include function definitions, global variable declarations, input declarations, and a global update block.

A.3 Global variables

Global variables, which include input variables, provide a mechanism to communicate information to all of the strands in the program. They are typically declared and initialized at the beginning of the program and may be updated in the global update block.Globals may be annotated as input variables, in which case they can be initialized from outside the Diderot program (either from the command-line in a standalone executable or using a library call).

A.4 Functions

Diderot allows the definition of user-defined functions at top level. There are two syntactic forms. For a function whose definition can be given as an expression, we write

```plaintext
function type Name ( parameters ) = expression;
```

whereas if the function implementation requires local variables and/or statements, we write

```plaintext
function type Name ( parameters ) { statements }
```

A.5 Strand definitions

Every Diderot program has a single strand definition, which has the form

```plaintext
strand Name ( parameters ) { 
    state variables 
    update { statements } 
    stabilize { statements } 
}
```

The (optional) strand parameters are used to pass instance-specific data to the strand initialization. The strand state variables hold the strand’s state; some variables may be annotated as output variables, which means that their final values are included in the result of the program. The strand state variables are initialized with the strand is created and are modified by the strand methods. Strands have one or two methods: the required update method that is invoked at each step and an optional stabilize method that is invoked when the strand stabilizes.

A.6 Global update block

Diderot programs may optionally include a global update block that is run at the end of each execution step. The code in this block may perform global reductions over the strands in the program and is also allowed to update global variables.

A.7 Global initially block

The global initially block is used to create the initial set of strands. Diderot uses a comprehension syntax, similar those of Haskell or Python, to define the initial set of strands. For example, the following code specifies a grid of initial ray positions:

```plaintext
initially { 
    RayCast( ui, vi ) | vi in 0..imgResV-1, ui in 0..imgResU-1 
}
```

When the strands are initialized as a grid, it implies that the strands will stabilize (i.e., they do not die). The grid structure is then preserved in the output.

Diderot also allows one to specify an initial collection of strands by using "( )", as the brackets around the comprehension (instead of "{ }"). In this case, the program’s output will be a one-dimension array of values for each output variable in a stable strand.

A.8 Statements

Statements in Diderot are a subset of C statements and include local-variable definitions, assignments, conditionals, and blocks that define a nested scope. In addition, there are several statement forms that are special to Diderot. The new statement is used to dynamically create a new strand. The stabilize and die statements terminate the calling strand; in the case of stabilize, the strand’s state is preserved and may contribute to the remaining computation and to the result. Lastly, Diderot has a looping statement form (foreach) that provides iteration over dynamic sequences.

A.9 Expressions

Diderot’s expression syntax is fairly standard, with infix binary operators, unary operators, function application (including field application), etc. Conditional expressions follow the Python syntax. We use Unicode characters for operators and literals to support traditional mathematical notation. Tensor values can be constructed from lower-order tensors using the notation \([\epsilon_1, \ldots, \epsilon_n]\). We also provide the special tensor literals identity\( d \) for the \( d \times d \) identity matrix, zeros\( \sigma \) for the zero tensor of the given shape, and nan\( \sigma \) for the tensor of the given shape initialized to NaN.

A.10 Operators and built-in functions

Diderot provides a rich set of unary and binary operations, as well as various built-in functions. These can be overloaded (\( e.g., + \) works on integers, tensors, fields, etc.) and polymorphic (\( e.g., + \) operates on tensors of any shape). A major use of overloading is to support lifted versions of tensor operations. We first present the overloaded operators and functions, along with the types at which they are defined. In this description, we use the type synonyms real, vec2, and vec3.

A.10.1 Overloaded operators and functions

**Comparison operators:** \(<,<=,>,>=\)

- int * int \(\rightarrow\) bool
- real + real \(\rightarrow\) bool

**Equality operators:** \(-=, !=\)
int + int → bool
real + real → bool
bool + bool → bool
string + string → string

Minimum and maximum (including reductions): min, max
int + int → int
real + real → real
real(1) + real(1) → real(1)

Unary negation: -
tensor(σ) → tensor(σ)
field(k)[d] → field(k)[d][σ]

Logical negation: not
bool → bool

Addition operators: +, -
tensor[σ] + tensor[σ] → tensor[σ]
tensor[σ] + field(k)[d] → field(k)[d][σ]
real + field(k)[d] → field(k)[d][σ]
real + field(k)[d] → field(k)[d][σ]

Scalar multiplication: *
tensor[σ] * tensor[σ] → tensor[σ]
tensor[σ] * field(k)[d] → field(k)[d][σ]
field(k)[d] * field(k)[d] → field(k)[d][σ]
field(k)[d] * tensor[σ] → field(k)[d][σ]

Scalar division: /
tensor[σ] / tensor[σ] → tensor[σ]
tensor[σ] / field(k)[d] → field(k)[d][σ]
field(k)[d] / field(k)[d] → field(k)[d][σ]
field(k)[d] / tensor[σ] → field(k)[d][σ]

Scalar exponentiation: ^
real * int → real
real * real → real

Inner product: •
tensor[σ1,d1,d2] • tensor[d1,d2,σ2] → tensor[σ1,σ2]
tensor[σ1,d1,d2] • field(k)[d1] • [d,d',σ2] → field(k)[d1] • [d,d',σ2]
field(k)[d1] • tensor[d1,d2,σ2] → field(k)[d1] • tensor[d1,d2,σ2]
field(k)[d1] • field(k)[d1] • tensor[d1,d2,σ2] → field(k)[d1] • field(k)[d1] • tensor[d1,d2,σ2]

Double-dot product: :
tensor[σ1,d1,d2] : tensor[d1,d2,σ2] → tensor[σ1,σ2]
tensor[σ1,d1,d2] : field(k)[d1] : [d1,d2,σ2] → field(k)[d1] : [d1,d2,σ2]
field(k)[d1] : tensor[d1,d2,σ2] → field(k)[d1] : tensor[d1,d2,σ2]
field(k)[d1] : field(k)[d1] : tensor[d1,d2,σ2] → field(k)[d1] : field(k)[d1] : tensor[d1,d2,σ2]

Cross product: x
vec2 • vec2 → real
vec3 • vec3 → vec3
field(k)[d][2] • field(k)[d][2] → real
field(k)[d][3] • field(k)[d][3] → field(k)[d][3]

Outer product: ⊗
tensor[d1] • tensor[d2] → tensor[d1,d2]
field(k)[d1] • field(k)[d2] → field(k)[d1,d2]

Convolution: ★
image(d)[σ] • kernel(k) → field(k)[d][σ]
kernel(k) • image(d)[σ] → field(k)[d][σ]

Gradient: V
field(k)[d+1][d] → field(k)[d][d]

Tensor derivative: V ⊙
field(k+1)[d] → field(k)[s][d]

Curl: V ×
field(k+1)[2] → field(k)[2]
field(k+1)[3] → field(k)[3]

Divergence: V •
field(k+1)[d] → field(k)[d]

Norm: ||
tensor[σ] → real
field(k)[d] → field(k)[d]

Normalization: normalize
tensor[σ] → tensor[σ]
field(k)[d] → field(k)[d]

Trace of a matrix: trace
tensor[d,d] → real
field(k)[d] → field(k)[d]

Transpose of a matrix: transp
tensor[d1,d2] → tensor[d2,d1]
field(k)[d1] → field(k)[d2]

Eigenvalues: evals
tensor[2,2] → real[2]
tensor[3,3] → real[3]

Eigenvectors: evecs
tensor[2,2] → vec2[2]
tensor[3,3] → vec3[3]

Determinant: det
tensor[2,2] → real
tensor[3,3] → real
field(k)[2,2] → field(k)[2]
field(k)[3,3] → field(k)[3]

Clamping: clamp
real * real * real → real
tensor[d] • tensor[d] → tensor[d]
Lerp: lerp

\[
\text{tensor}[\sigma] * \text{tensor}[\sigma] * \text{real} \rightarrow \text{tensor}[\sigma]
\]
\[
\text{tensor}[\sigma] * \text{tensor}[\sigma] * \text{real} + \text{real} + \text{real} + \text{real} \\
\rightarrow \text{tensor}[\sigma]
\]

Trigonometry: arccos, arcsin, cos, sin

\[
\text{real} \rightarrow \text{real}
\]
\[
\text{field}^{k}(d)[{} \rightarrow \text{field}^{k}(d)[{}]
\]

Sequence concatenation: @

\[
\tau * \tau{[]} \rightarrow \tau{[]}
\]
\[
\tau{[]} * \tau{} \rightarrow \tau{[]}
\]
\[
\tau{[]} * \tau{[]} \rightarrow \tau{[]}
\]

A.10.2 Other operators and functions

Pointwise vector multiplication: modulate

\[
\text{tensor}[d] * \text{tensor}[d] \rightarrow \text{tensor}[d]
\]

Testing the domain of a field: inside

\[
\text{tensor}[d] * \text{field}^{k}(d)[\sigma] \rightarrow \text{bool}
\]

Sequence length: length

\[
\tau{[]} \rightarrow \text{int}
\]

Boolean-sequence reductions: all, exists

\[
\text{bool}() \rightarrow \text{bool}
\]

Scalar-sequence reductions: mean, product, sum, variance

\[
\text{real}() \rightarrow \text{real}
\]

Loading images: image

\[
\text{string} \rightarrow \text{image}(d)[\sigma]
\]

Image dimensions: load

\[
\text{image}(d)[\sigma] \rightarrow \text{int}(d)
\]

Loading sequences: load

\[
\text{string} \rightarrow \tau[\sigma]{[]}
\]
B Embedding Diderot into an Application

In addition to compiling to a self-contained executable, a Diderot program may be compiled to a C library, which allows Diderot programs to be embedded in any language that supports calling C code. We demonstrate this here with the program that produced the dense isocontour sampling seen in Fig. 1(d). The complete Diderot code is below.

```diderot
// initialize the Diderot program
ISO_Initially (wrld);

// global initialization
ISO_Init (wrld);

// wrapper for output data
Nrrd *nout = nrrdNew();

// Main loop
while (ISO_NumActive(wrld) > 0) {
  // get and render the state
  if (ISO_Snapshot_pos(wrld, nout)) {
    // handle error...
  }
  // step the computation
  ISO_Run (wrld, 1);
}

// get and render final state
if (ISO_OutputGet_pos(wrld, nout)) {
  // handle error...
  Draw (nout);
  // shutdown the world
  ISO_Shutdown (wrld);
}
```

Lines 4–25 start the Diderot program. Line 5 allocates a new execution context, called a world. This context contains all runtime state of the program. Line 8 requests 8 cores for parallel execution, but only when the program was compiled for the SMP target. Line 12 allocates the computational resources needed to support the program’s execution (such as memory and threads). Lines 15 through 19 initialize those program input variables lacking defaults (the same ones set via command-line options above). Line 23 initializes the program, including allocating the initial set of strands. Line 25 allocates an Nrrd struct to contain the program output.

The while loop (lines 28–36) runs the program until all of the strands have stabilized or died. For each iteration of the loop, we grab (line 30) a snapshot of the program state (the union of all active and stable strands) and display it (line 33) with some Draw function, the details of which are not important here. For this program, which computes a collection of strands (due to the initially (...) in lines 30 and 31 of Diderot code above), the output allocated in nout by ISO_Snapshot_pos will be a two-dimensional 2–by–N array, where N is the number of output values, and the (faster) first axis is length 2 because the output variable pos is a vec2. Had this program initialized on grid of strands (with initially (...) the output would be a higher dimensional array, with one axis for each dimension of the grid, and an additional axis if the output variable was not real.

Each while loop iteration executes (line 35) one super-step of the Diderot program. It is also possible to run all strands to completion (without being able to access snapshots of intermediate state) with ISO_Run(wrld, 0). The while loop terminates when there are zero active strands (line 28). The final output of the program execution is grabbed by line 39 and displayed in line 42. The resources allocated by program initialization and execution are released by line 45.

The executable has command-line options to set all the input variables; values must be given for those lacking defaults. Inputs stepsMax and epsilon do have defaults (lines 6 and 7). The final “–n 8” option sets the number of cores for parallel execution, available when the program was compiled for the SMP target.

When compiling a Diderot program to a library, the compiler produces functions for initializing the program, setting the values of input variables, stepping through program execution, and getting program output. To avoid symbol name clashes, one specifies to the Diderot compiler a string, for example “ISO_”, which the compiler will use as a prefix for all symbols in the generated C library API. The C code below calls into the API generated from the Diderot program above, to replicate the effects of the invoking the stand-alone executable.

```diderot
// Create execution context
ISO_World_t *wrld = ISO_New ();

// initialize inputs
float cent[2] = {61.0, 40.7};
ISO_InVarSet_cent (wrld, cent);
ISO_InVarSet_hght (wrld, 53);
ISO_InVarSet_size0 (wrld, 50);
ISO_InVarSet_size1 (wrld, 30);
```

The stand-alone executable produced by compiling the program above can be run with:

```diderot
isofind -cent 61.40.7 -hght 53 -size0 50 -size1 30 -n 8
```

The executable has command-line options to set all the input variables; values must be given for those lacking defaults. Inputs stepsMax and epsilon do have defaults (lines 6 and 7). The final “–n 8” option sets the number of cores for parallel execution, available when the program was compiled for the SMP target.

When compiling a Diderot program to a library, the compiler produces functions for initializing the program, setting the values of input variables, stepping through program execution, and getting program output. To avoid symbol name clashes, one specifies to the Diderot compiler a string, for example “ISO_”, which the compiler will use as a prefix for all symbols in the generated C library API. The C code below calls into the API generated from the Diderot program above, to replicate the effects of the invoking the stand-alone executable.

```diderot
// Create execution context
ISO_World_t *wrld = ISO_New ();

// initialize inputs
float cent[2] = {61.0, 40.7};
ISO_InVarSet_cent (wrld, cent);
ISO_InVarSet_hght (wrld, 53);
ISO_InVarSet_size0 (wrld, 50);
ISO_InVarSet_size1 (wrld, 30);
```
C COMPARISON WITH HAND-CODED OPENCL

We show here comparisons between Diderot and hand-coded OpenCL code, for two simple volume renders: a maximum intensity projection (MIP), and isosurface rendered colormapped by the second derivative. These examples are somewhat favorable to OpenCL due to their simplicity compared to the previous examples in the paper, but the properties of the code, and changes required for a more sophisticated algorithm, nonetheless demonstrate benefits of Diderot.

1 // coefficients of piece-wise cubic bspln3
2 __constant float4 h[4][4] = {
3 { 1.333333f, 2.0f, 1.0f, 0.166667f }, // -2 .. -1
4 { 0.666667f, 0.0f, -1.0f, 0.5f }, // -1 .. 0
5 { 0.666667f, 0.0f, -1.0f, 0.5f }, // -1 .. 0
6 { 1.333333f, -2.0f, -1.0f, -0.166667f }, // 1 .. 2
7};
8 __constant int ha = 2; // half of support=4
9
10 #define FLOAT4(p) (float4)(p[0], p[1], p[2], p[3])
11
12 static inline float4 probeval(float4 volData, int sx, int sy, float4 volPos) {
13    float4 hf = convert_int4(volPos);
14    int4 n = convert_int4(nf);
15    float4 h = d + t * (c + t * (b + t * (a - 1))); // slowest
16    float4 hy = d + t * (c + t * (b + t * (a - 2))); // half of support=4
17    float4 hx = d + t * (c + t * (b + t * a));
18    float vx[4], vy[4];
19    for (int i = 0; i < 4; i++) {
20        vx[i] = (volPos.x + i - sx, volPos.y + i - sy, 0, 0);
21        vy[i] = (volPos.x + i - sx, volPos.y + i - sy, 0, 0);
22    }
23    float4 out = dot(FLOAT4(vx), hy);
24    return dot(FLOAT4(vy), hx);
25}

26 __constant float4 eye = FLOAT4(-240.538, 396.133, 64.6804f);
27 __constant float4 orig = FLOAT4(-47.5009, 48.5999, 71.5282f);
28 __constant float4 du = FLOAT4(0.170942, 0.146426, -0.0851239f);
29 __constant float4 dv = FLOAT4(0.0499616, 0.0720342, 0.224239f);
30 __constant float4 raystep = 0.3f;
31 __constant float4 lenmax = 130;
32 __kernel void kern(float4 global ushort * volData, float4 w12Pos, float4 12Grad, // (unused)
33    int imgSizeU, int imgSizeV, global float4 out) {
34    int ui = get_global_id(0), vi = get_global_id(1);
35    int sx = volSize[0];
36    int sy = volSize[1];
37    float4 pos0 = orig + ui*du + vi*dv;
38    float4 raydir = normalize(pos0 - eye); 0.0128;
39    float mup = 8;
40    for (float rlen = 0; rlen < lenmax; rlen += raystep) {
41        float4 pos = pos0 + rlen*raydir;
42        image<uchar4> * F = bspln3 @ image("vrthand.nrrd");
43        int imgSizeU;
44        int imgSizeV;
45        int ui, vi, int ui) {
46            vec3 pos3D = orig + ui*du + vi*dv;
47            vec3 raydir = normalize(pos0 - eye);
48            real len = 0;
49            output real min = 0;
50            update (ui, vi, vec3 pos = pos0 + len*raydir) {
51                if (inside(pos3D, F)) {
52                    min = max(F(pos3D), min);
53                }
54                vec3 rlen = raystep;
55                if (rlen >= lenmax) {
56                    stabilize;
57                }
58            }
59            
60            initially { raycast(ui, vi) | vi in 0..imgSizeV-1, 0..imgSizeU-1};
61        }
62        return dot(FLOAT4(vy), hx);
63    }
64    __constant float4 eye = FLOAT4(-240.538f, 396.133f, 64.6804f);
65    __constant float4 orig = FLOAT4(-47.5009f, 48.5999f, 71.5282f);
66    __constant float4 du = FLOAT4(0.170942f, 0.146426f, -0.0851239f);
67    __constant float4 dv = FLOAT4(0.0499616f, 0.0720342f, 0.224239f);
68    __constant float4 raystep = 0.3f;
69    __constant float4 lenmax = 130;
70    __kernel void kern(float4 global ushort * volData, float4 w12Pos, float4 12Grad, // (unused)
71    int imgSizeU, int imgSizeV, global float4 out) {
72    int ui = get_global_id(0), vi = get_global_id(1);
73    int sx = volSize[0];
74    int sy = volSize[1];
75    float4 pos0 = orig + ui*du + vi*dv;
76    float4 raydir = normalize(pos0 - eye); 0.0128;
77    float mup = 8;
78    for (float rlen = 0; rlen < lenmax; rlen += raystep) {
79        float4 pos = pos0 + rlen*raydir;
80        image<uchar4> * F = bspln3 @ image("vrthand.nrrd");
81        int imgSizeU;
82        int imgSizeV;
83        int ui, vi, int ui) {
84            vec3 pos3D = orig + ui*du + vi*dv;
85            float4 raydir = normalize(pos0 - eye);
86            
87    Above is hand-written OpenCL code for a simple ray-casting maximum intensity projection volume renderer. The code begins (lines 2-7) with a hard-coded definition of cubic B-spline kernel, in terms of its four piecewise cubic segments over $[-2,2]$. The probeval function (lines 12-42) implements convolution-based reconstruction with this kernel.

Fig. 12. 640×480 maximum intensity projection of right hand of Visible Human female CT scan.
The Diderot-generated OpenCL code is not has fast as the handwritten code, although it is more flexible. On the same platform as used for benchmarking (Sec. 6), the OpenCL code took on average 0.53 seconds to run (excluding compilation and disk I/O), while the Diderot code took on average 1.46 seconds. As noted in Sec. 6, our OpenCL implementation does not currently perform as well as might be hoped, work on it continues. On the other hand, as noted in Sec. 1, in creating Diderot we are not seeking to optimize execution time so much as human implementation time, with the long-term goal of assisting the rapid development and application of novel visualization algorithms.

The next rendering example illustrates this. Below is Diderot code for a kind of shaded pseudo-isosurface volume rendering. The rendering algorithm was designed to help assess whether an isosurface coincides with a material edge, considered as the zero-crossing of the second-directional derivative along the gradient direction. This is complementary to the Canny edge criterion visualized in Sec. 4.1, and requires only second (not third) derivatives.

```diderot
vec3 eye = [-240.538, 396.133, 64.6804];
vec2 orig = [-47.5009, 48.5995, 71.5282];
vec3 du = [0.176942, 0.146426, -0.0851239];
vec3 dv = [0.0492961, 0.0720342, 0.2242393];
real raystep = 0.1;
real lenna = 130;
real valIso = 1440;
field3d$h[3]$ F = bspln3 @ image ("vfhand.nrrd") - valIso;
input int imgsizedU;
input int imgsizedV;
real ValToI = 150;
real isoThick = 3;
vec3 light = normalize([-1.2,-1,1]);
vec3 gray = 0.7*(1,1,1);
function real atxf(real v, real gg) {
return clamp(0, 1, 1 - |vv|/(gg*isoThick));
}

raycast (int vi, int ui) |
vec3 pos0 = orig + vi*du + ui*dv;
vec3 raydir = norm = -(pos0 - eye);
real rlen = 0;
output vec rgb = [0,0,0];
vec3 rgb = [0,0,0];
real transp = 1;
update |
vec3 pos0 = pos - rlen*raydir;
if (inside (pos0, F) && F(pos) <= valToI) {
real alpha = atxf(F(pos0), F(valPos));
if (alpha > 0) {
vec4 norm = normalize(F(pos0));
real lit = lerp(0.1, -1, norm/lit, 1,2);" normal/lit, 1,2);" normal/lit, 1,2);" normal/lit, 1,2);" normal/lit, 1,2);"
real addn = norm + F(pos0) + norm/(600);
vec3 col = lerp(gray, [1.0, 0.0, 0.7]) - (addn);
if (add <= 0) else
lerp(gray, [0.5, 0.3, 0.3], (addn));
rgb = transp*alpha*lit*col;
transp = 1 + alpha;
}
rlen += raystep;
if (rlen > lenna) |
stabilise;
}

stabilise |
real a = 1 - transp;
if (a > 0) |
rgb = [rgb[0]/a, rgb[1]/a, rgb[2]/a, a];
}
}

initially |
raycast(vi, ui) @ vi in 0..imgsizeV-1;
ui in 0..imgsizedV-1;" normal/lit, 1,2);" normal/lit, 1,2);" normal/lit, 1,2);" normal/lit, 1,2);"

The key lines are 36-39: the Hessian \( \nabla V \otimes \nabla F(\text{pos}) \) is contracted (by \( \cdot \)) on both sides by the unit-length isosurface normal \( \text{normalize}(\nabla V \otimes F(\text{pos})) \) to find the second directional derivative add of \( F \) along \( \text{norm} \). Depending on the sign of add, a per-sample color \( \text{col} \) is found by lerping between gray and magenta (negative) or green (positive). Assuming materials of interest are more dense than the background, if the isovalue is too low, the isosurface is outside the material boundary and hence in a region of positive second derivative. Conversely, isovalue too high will put the isosurface within a negative second derivative.

This is precisely what is seen in the resulting rendering, in Fig. 13. The isosurface is outside the dense cortical bone surface in the mid-dle of the phalanges, but inside the thinner bone surface at the joints. Four lines 36-39 of code make an otherwise generic isosurface volume rendering into an informative illustration of the mathematical properties of isosurfaces and the bone surface. The Diderot programmer is unburdened by the complexity associated with correctly computing \( \nabla \nabla F(\text{pos}) \) in world space. For this more complex rendering task, however, the performance hit from Diderot (relative to hand-written OpenCL code) is smaller than in the MIP example. The hand-written OpenCL code took on average 1.84 seconds, while the Diderot code took on average 3.30 seconds.

The OpenCL code is below. The computational kernel (starting line 194) is about as legible as the Diderot update code, though the double contraction of the Hessian (lines 232-234) is not as clear. Also, whereas the Diderot compiler can easily optimize repetitions of \( \nabla F(\text{pos}) \) with common subexpression elimination, the OpenCL programmer is more likely to call functions like \( \text{probeGrad} \) once (line 224) and save the results in a suggestively named variable. Given the mathematical limitations of OpenCL, is to hard to correctly implement \( \text{probeGrad} \) (lines 44-86) for reconstructing the world-space gradient, and \( \text{probeHess} \) (lines 88-166) for the Hessian, especially the conversion of the index-space Hessian to world-space (lines 144-163). Even with these utility functions written for scalar data, however, the OpenCL programmer is no closer to having code for evaluating \( \nabla (\nabla F) \) let alone \( \nabla \nabla (\nabla \nabla F) \) for rendering Canny edges, which Diderot generated automatically. Additional OpenCL code would be needed for doing convolution-based reconstruction of vector- and tensor-valued data. Handling derivatives in vector and tensor fields (including conversions from index to world space) would be challenging even for the most meticulous programmer. We suggest that this is exactly the kind of tedious error-prone work that should be automated by the compiler of a high-level language like Diderot.

![Fig. 13. 640x480 rendering of isosurface at 1440 of right hand of Visible Human female CT scan, colored by second directional derivative (green: positive, gray: zero, magenta: negative).](image-url)
float4 h0x = d + t * (c + t * (d + t * a));
float4 h1x = d + t * (c + t * (d + t * a));
float4 h0y = t = (float4) (f.x * 1, f.x, f.x - 1, f.x - 2);
float4 h1y = t = (float4) (f.x * 1, f.x, f.x - 1, f.x - 2);
float4 h0z = d + t * (c + t * (d + t * a));

float vx[4], vy[4]; // grid support
for(int k = 1-hs; k <= hs; k++) {
    for(int j = 1-hs; j <= hs; j++) { // medium
        int index = n.x - 1 + sx*(n.y + j) + sy*(n.z + k);
        float4 v = FLOAT4(volData + index);
        vy[khs-1] = dot(v, vx);
    }
}

static inline float3 probeGrad (float4 12wGrad,
    int sx, int sy, float4 VolPos) {
    float4 nf, t;
    float4 d = (float4) (h[3][0],h[2][0],h[1][0],h[0][0]);
    float4 c = (float4) (h[3][1],h[2][1],h[1][1],h[0][1]);
    float4 b = (float4) (h[3][2],h[2][2],h[1][2],h[0][2]);
    float4 a = (float4) (h[3][3],h[2][3],h[1][3],h[0][3]);
    float f = modf(VolPos.xn); nf = t = (float4) (f.x * 1, f.x, f.x - 1, f.x - 2);
    float4 h0x = d + t * (c + t * (d + t * a));
    float4 h1x = d + t * (c + t * (d + t * a));
    float4 h0y = t = (float4) (f.y * 1, f.y, f.y - 1, f.y - 2);
    float4 h1y = t = (float4) (f.y * 1, f.y, f.y - 1, f.y - 2);
    float4 h0z = d + t * (c + t * (d + t * a));
    float4 h1z = d + t * (c + t * (d + t * a));
    float4 h0y = h0y + h0z;
    float4 h1y = h1y + h1z;
    float4 h0z = h0z + h1y;
    float4 h1z = h1z + h0y;
    float3 grad = (float3) (dot(FLOAT4(vx[0y]), h0z), dot(FLOAT4(vx[1y]), h1z), dot(FLOAT4(vx[2y]), h1y), dot(FLOAT4(vx[3y]), h0y));
}


t = (float4) (f.x * 1, f.x, f.x - 1, f.x - 2);

// convert hessian to world space, 1st index
ret.s0123 = (float4) (dot(thes0, imgSizeW), dot(thes1, imgSizeW), dot(thes2, imgSizeW), dot(thes3, imgSizeW));

// convert hessian to world space, 2nd index
ret.s4568 = (float4) (dot(thes0, imgSizeW), dot(thes1, imgSizeW), dot(thes2, imgSizeW), dot(thes3, imgSizeW));

// convert hessian to world space, 3rd index
ret.s8945 = (float4) (dot(thes0, imgSizeW), dot(thes1, imgSizeW), dot(thes2, imgSizeW), dot(thes3, imgSizeW));

// convert hessian to world space, 4th index
ret.scdet = (float4) (0.0f, 0.0f, 0.0f, 1.0f);
return ret;

// constant float4 eye = (float4) (-240.538f, 396.1337f, 64.68045f, 1.0f);
// constant float4 orig = (float4) (-47.50095f, 48.59995f, 71.5282f, 1.0f);
// constant float4 du = (float4) (0.170942f, 0.146426f, -0.0851239f, 0.0f);
// constant float4 dv = (float4) (0.049961f, 0.0723042f, 0.2243295f, 0.0f);
// constant float raystep = 0.1f;
// constant float imageMax = 130;
// constant float vailso = 1440;
// constant float vailto = 150;
// constant float isoThick = 3;
// constant float3 light = (float3) (-0.4082f, 0.8165f, -0.0802f);
// constant float3 gray = (float3) (0.7);

static inline float4 atxf (float v, float g) {
    float av = fabs(v - vailso);
    return clamp(1 - av/(gg*isoThick) , 0.0f, 1.0f);
}

// kernel void kern (global ushort * volData,
//    float64 * vol4Pos,
//    int imgSz, int imgSzV, int s4,
//    global float * out)
//    global float * out)
int ui = get_global_id(0), vi = get_global_id(1);
int sx = volSize[0];
int sy = volSize[1];
int sz = volSize[2];

float4 pos0 = orig + ui*du + vi*dv;
float3 raydir = normalize(pos0 - eye).x0yz;
float norm = 0;

float4 t = (float4) (f.x * 1, f.x, f.x - 1, f.x - 2);

float4 h0x = d + t * (c + t * (d + t * a));
float4 h1x = d + t * (c + t * (d + t * a));
float4 h0y = t = (float4) (f.y * 1, f.y, f.y - 1, f.y - 2);
float4 h1y = t = (float4) (f.y * 1, f.y, f.y - 1, f.y - 2);
float4 h0z = d + t * (c + t * (d + t * a));
float4 h1z = d + t * (c + t * (d + t * a));
float4 h0y = h0y + h0z;
float4 h1y = h1y + h1z;
float4 h0z = h0z + h1y;
float4 h1z = h1z + h0y;
float4 h0y = h0y + h0z;
float4 h1y = h1y + h1z;
float4 h0z = h0z + h1y;
float4 h1z = h1z + h0y;
float3 grad = (float3) (dot(FLOAT4(vx[0y]), h0z), dot(FLOAT4(vx[1y]), h1z), dot(FLOAT4(vx[2y]), h1y), dot(FLOAT4(vx[3y]), h0y));
float3 col = ((sdd < 0) ? mix(gray, (float3)(1.0, 0.0, 0.7), -sdd) : mix(gray, (float3)(0.3, 1.0, 0.3), sdd));
rgb += transp*alpha*lit*col;
transp *= 1 - alpha;
}

if (aa > 0) {
opix[0] = rgb.s0/aa;
opix[1] = rgb.s1/aa;
opix[2] = rgb.s2/aa;
opix[3] = aa;
} else {
opix[0] = 0;
opix[1] = 0;
opix[2] = 0;
opix[3] = 0;
}