https://doi.org/10.2352/ISSN.2470-1173.2017.1.VDA-396

Closest Point Sparse Octree for Surface Flow Visualization

Mark Kim and Charles Hansen; Scientific Computing and Imaging Institute at the University of Utah; Salt Lake City, UT. USA

Abstract

As datasets continue to increase in size and complexity, new techniques are required to visualize surface flow effectively. In this work, we introduce a novel technique for visualizing flow on arbitrary surface meshes. This new method utilizes the closest point method (CPM), an embedding technique for solving partial differential equations (PDE) on surfaces. The CPM operates by extending values off the surface into the grid and using standard three dimensional PDE stencils to solve embedded two dimensional surface problems. To adapt unsteady flow visualization for the CPM, unsteady flow line integral convolution (UFLIC) is applied in three dimensions to the embedded surface in the grid to visualize flow on an arbitrary surface.

To address the increased size and complexity of datasets, we introduce the closest point sparse octree to efficiently represent an embedded surface. By constructing a closest point sparse octree, complex surfaces can be represented in a memory efficient manner. Further, various techniques, such as a Laplacian filter, can be applied more easily to the embedded surface because of the CPM. Finally, the memory efficiency of our new sparse octree approach allows grids to be constructed up to $8,192^3$ *in size on a GPU with 12GB of RAM.*

Visualizing vector fields is important and pervasive in a myriad of fields such as computational fluid dynamics, aerospace, and weather simulation. Many techniques have been developed for vector field visualization, such as Line Integral Convolution (LIC) [5]. LIC, and its variants, is a popular and fundamental vector field visualization method because of its simplicity and portability.

Because of its similarity to oil flow visualization in aeronautics, LIC can also be appealing for vector field visualization on surfaces. There are two different methodologies for LIC on surfaces: image-space methods and surface parameterization methods. Image-space methods project the visible surface geometry and velocity field into the image space and LIC is applied in the image space [16, 32]. By depositing values of just the visible portions of the surface, the visualization is highly interactive due to the parallel nature of LIC on a GPU. This interactivity comes at a cost though: there can be artifacts from altering the camera position which can be noticed around silhouette edges. Further, self-occluded areas can be incorrect because the area under the occluded area are not processed.

In contrast to image-space techniques, a surface can be parameterized and flow visualization techniques applied in the parameter space. The state of the art in parameter space techniques utilizes the closest point embedding to apply UFLIC to surfaces [13]. Unfortunately, this nearly-interactive scheme does not scale well as the surface requires a higher resolution grid which drastically increases memory requirements.

In this paper we present a new method for visualizing flow utilizing a parameterized surface. In a previous paper, we introduced LIC using a closest point embedding [13]. This new research moves beyond just an embedding to use the closest point method for the surface flow visualization. As datasets continue to increase in size, new techniques are needed for effective surface flow visualization. Our approach utilizes a GPU-based streaming, sparse octree to scale the parameterized representation up to $8,192^3$. Further, we adapt LIC and the sparse octree to the closest point method, a technique for solving PDEs on embedded surfaces [25]. By using the closest point method, other partial differential equation-based flow visualization techniques, such as reaction-diffusion, could be applied to as well.

Our contributions are:

- Introduce the closest point method for surface flow visualization.
- Introduce a sparse octree closest point construction technique.
- Apply unsteady flow line integral convolution (UFLIC) to the closest point method for surface flow visualization.

To perform the surface flow visualization, a sparse closest point embedding is constructed by converting the triangular mesh into a sparse three-dimensional closest point octree. Then, unsteady flow line integral convolution, or UFLIC, carries a noise field through the three-dimensional sparse closest point grid by extending the values off the surface into a surrounding grid and applying UFLIC in three dimensions.

Related Works

We review the related works. First, flow on surfaces is reviewed. Then the works related to the closest point method are covered. Finally, the sparse octree construction is reviewed.

Flow on surfaces

A thorough review of vector field visualization is beyond the scope of this paper. We limit the review to relevant work on surface-based flow visualization and refers readers to [4] and [8] for a more thorough overview of flow visualization and surface flow visualization, respectively.

Surface Flow Visualization

Image-space surface LIC methods were introduced simultaneously by Laramee et al. [16] and van Wijk [32] (ISA and IBFVS, respectively). These are extensions of Image Based Flow Visualization [31], or IBFV, to surface flow visualization by utilizing the GPU perform LIC in image space. The IBFV method uses a white noise texture that is bent by the vector field and then blended with other white noise textures over time. This technique is very efficient on the GPU. Both the ISA and IBFVS extend IBVF by projecting the velocity field embedded in the surface geometry into the image space and bending and blending the tex-

tures in image space. Image-based methods are very efficient for surfaces with the normal inherent drawback of image based methods. To enhance the coherency of the output, the noise texture is mipmapped and the triangle-texture mapping is altered [11]. These adjustment do not solve the inherent problem of correct surface occlusion nor allow the use of other unsteady flow techniques such as dye advection [18, 12].

In contrast to image space approaches, surfaces can be parameterized and the flow visualization done in the parameter space. Parameterized surface flow visualization was first introduced by Forssell et al. [9]. This LIC-based approach generated the visualization in parameter space, but generally was not distortion free. Battke et al. performed LIC on a tessellated surface in the local coordinate space of the triangle [3]. Unfortunately, the triangulation needed to be good, thus limiting its usefulness.

Flow Charts is another parameterized unsteady flow visualization technique for surfaces [19]. The triangular mesh is decomposed into patches using a particle system, which are then packed into textures. Three particle advection schemes for dense flow visualization, GPU Line Integral Convolution, Unsteady Flow Advection- Convolution and level-set dye advection, are used to visualize the vector field on the texture [17, 34, 33]. The texture is then mapped onto the surface during rendering. While Flow Charts allows for multiple visualization types, it has the following drawback: the pre-processing step to decompose the mesh with a particle system is very time-consuming and does not scale well. To speed-up the parameterization to near interactive speeds, Kim et al. introduced the closest point embedding, a fast parameterization that performs ULIC at near interactive rate [13]. Particles are placed on the surface and then advected according to the velocity field. Once advected, the particles are re-projected back onto the surface and their value is saved into on to the grid. This advection/projection scheme is continued until the particle has traveled a pre-determined length.

Closest Point Method

Ruuth and Merriman introduced the closest point method (CPM) as an embedding surface for solving partial differential equations (PDEs) [25]. The *CPM* allows for unmodified $\mathbb{R}^3$ differential operators to replace intrinsic surface operators normally used to solve PDEs on surfaces by extending values on the surface into the grid. Macdonald and Ruuth followed up the explicit time step with an implicit time step, as well as evolving a level-set on a surface [21, 20] and März and Macdonald followed up the works of Macdonald and Ruuth with proofs for the principles of the method [22].

There have been numerous applications of the *CPM* to surface problems. Tian et al. used the *CPM* for segmentation on a surface [30] and Hong et al. applied the *CPM* to the level-set equation to simulate fire on a surface [10]. Auer et al. used the closest point method to solve the Navier-Stokes equations on surfaces and introduced a higher order interpolant for creases [1]. Finally, Demir and Westermann recently used the closest point for a smoother approximation of an octree surface representation with ray casting [7]. Although designed for processing arbitrarily large meshes, the octree construction would not scale to the sizes necessary for the datasets we use.

Sparse Octree

A comprehensive overview of fast sparse octree voxelization is outside the purview of this paper. We refer the reader to Laine [14] for more detailed analysis.

Recently, there have been numerous fast, sparse GPU voxelization for rendering systems proposed. GigaVoxels, introduced by Crassin et al. [6], renders large volumetric datasets depending on viewpoint and adaptive data representation. Laine and Karras [15] are also rendering based, using a slice-based approach to construct a top-down tree. Schwarz and Seidel [28] replaced the 2D rasterization approach previously used with a set of "3D rasterizers" approach. This gave a more flexible scheme by reducing some of the redundant per-triangle processing.

On the other hand, Baert et al. [2] proposed a CPU out-of-core sparse voxelization approach. Although not as fast as previous implementations, it is the only method that is not bound by the available memory. Recently, this was extended to the GPU by Pätzold and Kolb [23] and achieved moderate speed-ups. The Schwarz and Seidel [28] is up to two orders of magnitude faster than [2], but it is not out an out-of-core method and may not have scaled to a high enough resolution for our purposes.

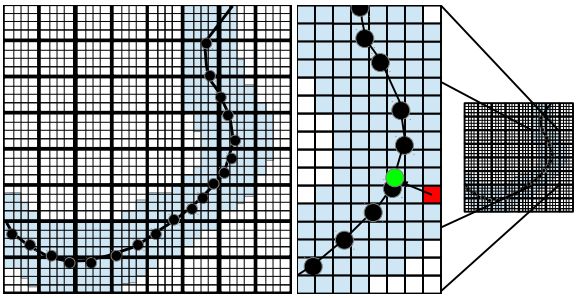
Embedding the Surface

To maintain as much flexibility as possible, we implement the sparse voxelization octree approach similar to Baert et al. [2] on the GPU. The closest point method is extended to use this data structure on the GPU. The sparse voxelization algorithm is a bottom-up sparse voxelization approach that proceeds in two steps: the voxelization and the sparse octree construction. Further, this is a hybrid approach where the voxelization and closest point embedding process is implemented in CUDA, whereas the sparse octree construction is on the CPU. The first phase inputs a triangular mesh and generates an intermediate sparse closest point grid using Morton order. The second phase produces a sparse octree through a streaming process using Morton order.

Sparse Closest Point Grid

The closest point grid is constructed from a surface mesh with the velocity field at the vertices of the mesh. Fig. 1a is a two-dimensional grid, where the blue cells are close to the surface and contain the closest point to the surface. The white cells are outside of the narrow band around the surface and therefore are excluded from the sparse octree. The closest point embedding stores the location on the surface that is nearest to the cell. Using Fig. 1b as an example, the cell at *(23,14)* is colored red, and the closest location on the surface to the cell is colored green. The value stored in the closest point embedding at the cell *(23,14)* is *(21.3,14.8)*, which is the closest surface point (green circle) to the red cell.

Construction of the closest point octree is as follows. The whole octree grid is decomposed into subgrids because the grid memory increases exponentially as the grid size increases. Then, for each grid cell and each triangle near the grid cell, a count of the number of triangles near the grid cell is computed. This count is needed to construct an array of triangles that are near to a grid cell. For each triangle in the subgrid, an axis aligned bounding box (AABB) is determined. Then, the AABB is expanded by a user-defined offset. In practice, to balance performance and accuracy the offset is set to 3. Then, for each grid cell in the AABB,



(a) A piecewise curve embedded into a grid.

(b) A subsection of the grid showing a closest point to the surface.

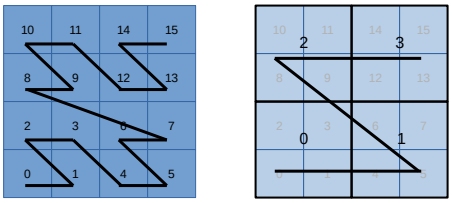
Figure 1: Figs (a) and (b) are two-dimensional examples of the closest point embedding. For all figures, the cells marked as close to the surface are colored blue, and cells far away from the surface are colored white. Cells colored blue are stored in the sparse octree, whereas cells colored white are discarded to save memory. Fig. (a) is an example surface, a curve embedded in a 24x24 2D grid. Fig. (b) displays part of the grid from (a) with an example of the closest point to the surface shown, where the red cell is at the fine grid position, *(23,14)* , the projection is visualized with an arrow, and the surface location (the green point), is at *(21.3,14.8)*.

the closest point to the grid cell on the triangle is computed. If the distance from the closest point to the grid cell is less than a user-defined value, *radius*, then a counter is incremented with *atomicInc* in *CUDA* because other triangles may also be near the grid cell. In practice, $radius = 5$. Next, an exclusive scan is performed on the grid cell counts, which gives us an index for each grid cell to have its own subarray of triangles. Further, it also computes the total number of triangles to cells needed, and a new array is constructed to store the triangle to cells.

After an array is created for storing lists of triangles close to grid cells, the array is filled in parallel. For each triangle in the subgrid and for each grid cell in its expanded axis-aligned bounding box (AABB), the triangle is stored in the triangles-to-cell array. Finally, for each grid cell that has a triangle near it, and for each triangle near it, the closest point is computed and if this is closer than previous triangles, it is stored. The velocity field is stored into its own sparse grid in a similar manner.

To compute the point on a triangular mesh closest to the grid cell, for every triangle in the grid, the triangle is translated and rotated such that one vertex is at the origin, and the two other vertices are in a coordinate plane. This translation and rotation transforms finding the closest point into a two-dimensional problem, where solving for the location in two dimensions gives seven regions where the projected grid vertex can lie [27].

To ensure scalability of the closest point construction, the grid is subdivided into subgrids depending on the amount of memory on the GPU. The number of partitions required is determined by the amount of memory needed to store a Morton code (8 bytes), a closest point (12 bytes), and a velocity vector (12 bytes) for each grid cell in the subcell (in the worst case), plus an integer (4 bytes) per grid cell to count the number of triangles that are near the cell.



(a) Highest level.

(b) Level below the highest.

Figure 2: A two-dimensional example of Morton order and its hierarchy. (a) is the highest level Morton order, and (b) is a coarser Morton order.

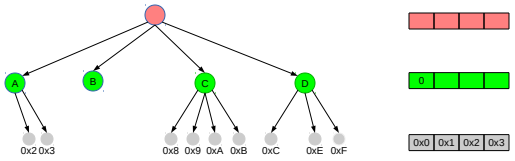
Morton Order

Morton order, or z-order (Fig. 2), is a multidimension to one dimension mapping that maintains locality. It is a hierarchical ordering such that the Morton order for a high level of the tree (Fig. 2a) is congruent to the Morton order of a lower level (Fig. 2b) of the octree. The purpose of using Morton codes for the octree construction is that Morton order allows a bottom-up construction. Further, Morton order makes it easier to divide the work into separate "queues," where there is one queue for each level of the octree.

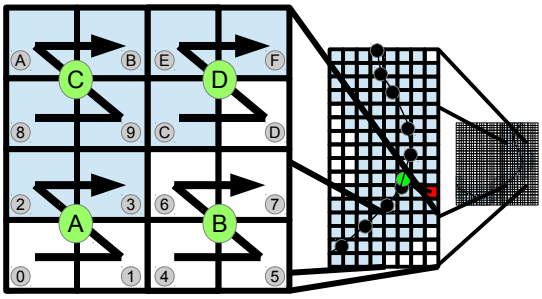
To construct the Morton code, the three-dimensional grid cell coordinate is stored interleaved in a 64-bit unsigned integer (Algorithm 1). To interleave the bits, for each bit b at position i in a grid cell coordinate c, a mask is created $(1 << i)$ and applied to that coordinate with a bitwise conjunction. Then, it is bitshifted by twice the bit position i, and the value is applied to the output with a bitwise disjunction. This procedure is carried out for each dimension of the grid cell coordinate. For instance, the coordinate $(23,6,14)$ is $(10101,00110,01110)$ in binary and interleaved 001100111110001 or the 6641st cell in the Morton order.

Algorithm 1 Interleaved Morton encode where the input is a grid cell coordinate, `(x,y,z)`, the output is the Morton code, *mc* and << is the left bitshift.

$mc \leftarrow 0$
$i \leftarrow 0$
while $i < 21$ **do**
  $mc \leftarrow mc \vee (x \wedge (1 << i) << i \times 2)$
  $i \leftarrow i + 1$
end while
$i \leftarrow 0$
while $i < 21$ **do**
  $mc \leftarrow mc \vee (y \wedge (1 << i) << i \times 2 + 1)$
  $i \leftarrow i + 1$
end while
$i \leftarrow 0$
while $i < 21$ **do**
  $mc \leftarrow mc \vee (z \wedge (1 << i) << i \times 2 + 2)$
  $i \leftarrow i + 1$
end while



(a) Sparse octree construction



(b) Sparse octree example

Figure 3: Continuing with the embedded piecewise curve example from Fig. 1, a 4×4 two-dimensional subgrid is used as an example to construct a sparse octree in (b). The cells are labeled *0x0* to *0xF* in Morton order. In (a) $0x0$ to $0x3$ are in the grey level and the parent-child relationship is recorded in the green level. In this example, only $0x2$ and $0x3$ are leaf nodes that exist in the closest point grid. The nodes $0x0$ and $0x1$ are empty keys.

Sparse Octree Construction

The sparse octree construction is as follows. Given a sorted-order Morton key list of occupied cells from the closest point construction , for each Morton key, place it in the queue at the highest level (leaf level) of the tree. Continue filling the highest level queue with Morton keys from the list of occupied cells or empty keys until the queue is full. Once the leaf level queue is full, a parent node is created in a queue at the second highest level and the parent-child relationship is recorded. The lowest level queue is reused for the next set of leaf nodes. This parent-child relationship recording is recursively done for each queue of the tree, until the tree is completed. Note, if a queue is filled with empty keys, then the queue can be skipped and a key inserted at the parent queue. This procedure makes for an efficient empty key skipping technique.

An example of the sparse octree construction is given in two-dimensions in Fig.3. At the highest level of the octree, Morton keys $0x2$ to $0x3$, along with empty keys $0x0$ and $0x1$, are placed in the queue. Then, a parent-child relationship is recorded at the parent node, *A,* on the green level and the queue is cleared. Then, empty keys $0x4$ through $0x7$, the parent node *B* is created and recorded in the queue at the green level, and the queue at the gray level is cleared. Then, the Morton keys $0x8$ through $0xB$ are placed in the queue, and the parent node is created in the queue at the green level. Finally, $0xC$, $0xE$ and $0xF$ Morton keys, with the empty key $0xD$, are copied to the queue. The parent node is then created in the green queue. Since the queue is done, the queue at the red level records the parent-child relation between the red and green levels. Fig. 3 is an example with the queue stopped after the parent-child relationship is recorded for the green level *A*.

Using the Closest Point Octree

Once the triangular mesh is converted to a sparse closest point octree, locating a cell now requires a tree-traversal of $O(log(n))$ time. Given a point within the domain of the closest point grid, the search starts at the root node. For each level in the tree, find the child node that encapsulates the point. This search continues down each level until either an empty node is reached or the leaf node is found.

Although the cost of any lookup is *O(log(n))* with the octree, this search can limit performance for three-dimensional stencil operations such as Laplacian or linear interpolation. Therefore, if there is enough memory on the GPU, a neighborhood index is constructed for each grid cell by doing a tree-traversal on each neighbor grid cell and stored in a neighborhood lookup.

Flow Visualization With the Closest Point Method

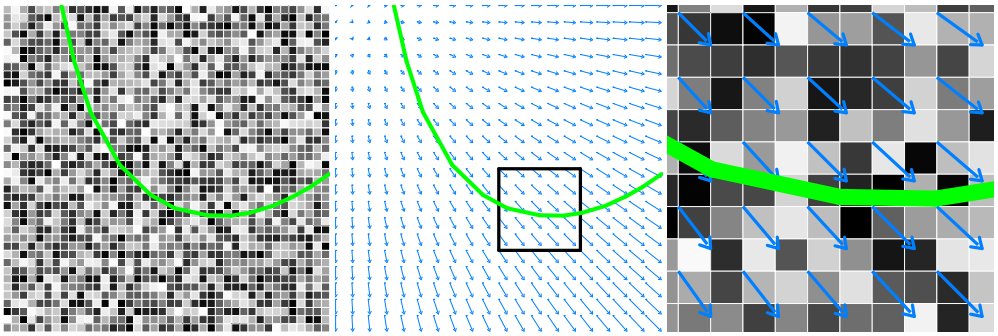
To demonstrate the effectiveness of the closest point method for flow visualization, we apply the unsteady flow line integral convolution, or UFLIC, to visualize surface flow. In this section, we describe usage of the UFLIC on the surface as well as the visualization of the surface flow.

Unsteady Flow Line Integral Convolution (UFLIC) is a technique to visualize unsteady flow [29]. In this scheme, particles are released from the center of every pixel and are advected forward, depositing their scalar value along the pathline. Once the advection and depositing are completed, the accumulated values are normalized, filtered, and jittered, creating the flow visualization.

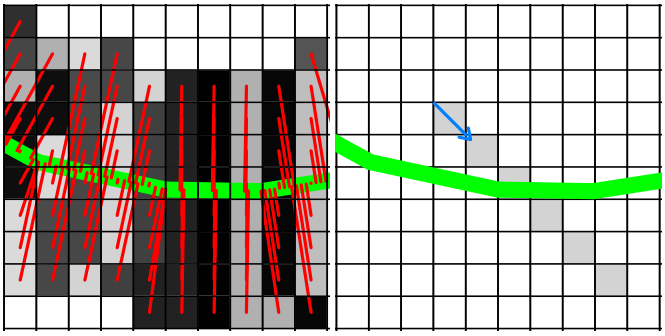
UFLIC

The closest point sparse octree is used to produce and visualize the UFLIC on the surface. Initially, given the closest point and velocity grid, a UFLIC sparse grid of the same size is filled with random noise, similar to how a two-dimensional UFLIC is initialized. Once the noise grid is constructed, the values on the surface are extended from the surface into the surrounding extension grid in the *extension phase*. To extend the surface values into the surrounding grid cells, for each grid cell, linearly interpolate the values around the closest point of the grid cell. Then the interpolated value is stored in the grid cell in the extension grid.

Once the white noise has been extended into the extension grid, for each cell, a point is placed at the center of the cell and stores the value of the initial grid cell. As the particles are advected through the grid, their initial values are accumulated in the UFLIC grid using a three-dimensional Bresenham line drawing algorithm. Once the advection process is complete, the field



(a) 40×40 noise grid. (b) 20×20 velocity field. (c) Zoomed into noise grid.



(d) Extending the values on the surface into the grid. (e) Applying UFLIC.

Figure 4: Fig. (a) through (e) are a two-dimensional example of the UFLIC with a one-dimensional embedded curve. Fig. (a) is a 40×40 noise grid, and Fig. (b) is a vector field, but 20×20 to reduce visual clutter. Fig. (b) also has a zoomed in portion for Figs. (c) and (d). Fig. (c) shows the extension phase of the closest point method. The values have been interpolated into the UFLIC grid. Finally, a single example is given in Fig. (e), where a value that was interpolated from the surface is then deposited back onto the surface with UFLIC.

is normalized, sharpened with a three-dimensional laplacian operator, and jittered. To visualize the surface, a parametric CPU raycaster is implemented [24].

An example of the extension and application of UFLIC is shown in Fig. 4. Fig. 4a and 4b are a 40×40 noise grid and a 20×20 velocity field, respectively. The velocity field is down sampled to reduce visual clutter. Fig. 4c shows the zoomed in region of Fig. 4b combined with the noise of 4a. Fig. 4d extends the surface values into the extension grid, where the closest point to a grid cell is shown with a red line. Finally, UFLIC is applied to the two-dimensional extended grid in Fig. 4e, but only a single particle advection is shown. The value that is deposited onto the UFLIC grid was interpolated off the surface in the extension phase.

After the particles are advected through the surface, the UFLIC grid is normalized and a standard three-dimensional laplacian filter is applied to all the cells. Then, the grid is clamped and jittered to prepare for the next iteration of UFLIC.

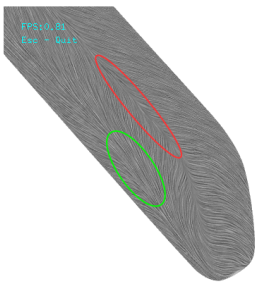
Results and Discussion

To validate the unsteady flow line integral convolution (UFLIC) on a surface using the closest point method, a visual comparison is performed between our technique and a previous unsteady surface flow visualization methods: the Closest Point Embedding [13]. Then, to demonstrate the closest point sparse octree performs and scales well, three datasets are used with varying grid sizes.

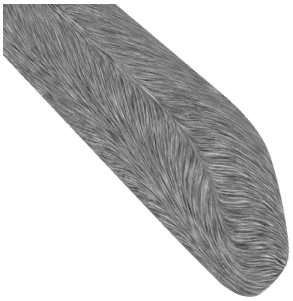
Validation

To demonstrate the UFLIC with the CPSO, two datasets are used: the ICE train and the F6 plane datasets (Figs. 5 and 6, respectively). An important goal is that the closest point sparse octree (CPSO) has comparable results to previous parametric unsteady flow surface visualization techniques: the Closest Point Embedding (CPE) [13]. A figure is provided for each dataset using CPE, as well as the CPSO, for comparison purposes. Both the ICE train and the F6 airliner are voxelized with a grid size of 1024^3, and the delta wing vortex bubble dataset is voxelized with a grid size of 8192^3. The grid size of 1024^3 for the ICE Train and F6 airliner was chosen because they are visually similar to the CPE UFLIC. However, the grid size of 8192^3 for the delta wing vortex bubble was chosen because it is the resolution that accurately represents the surface. The delta wing vortex bubble is a complex integral surface that tightly wraps around itself, and in some regions the surface is very close to itself. Therefore, a refined sparse octree, with a grid size of 8192^3, is needed to correctly represent the surface with the CPSO and to apply UFLIC properly.

The ICE train (Fig. 5) is a simulation of a high speed train traveling at 250 km/h with wind blowing at a 30 degree angle. The wind creates a drop in pressure, generating separation and

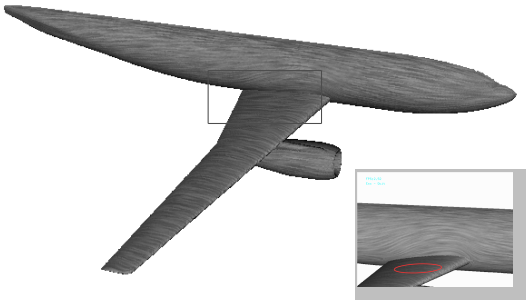


(a) The CPSO ICE Train

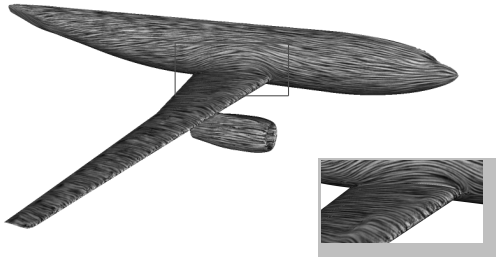


(b) The CPE ICE Train

Figure 5: The ICE train visualized with UFLIC with the closest point sparse octree and UFLIC (Fig. (a)) and the closest point embedding (Fig. (b)).



(a) The CPSO Airliner



(b) The CPE Airliner

Figure 6: The airliner (F6) dataset visualized with UFLIC and the closest point sparse octree (Fig. (a)) and the closest point embedding (Fig. (b)).

attachment flow patterns, which can be seen on the surface in Fig. 5a. Shear stress is shown on the airliner (F6) dataset, which is in Fig. 6a.

When generated with the Closest Point Embedding and the CPSO, both datasets are visually similar. For the ICE Train in Fig. 5, the separation (highlighted by the red circle) and attachment (highlighted by the green circle) flow patterns can be seen in both procedures. With the F6 dataset in Fig. 6, the shear stress from the wind (highlighted with a red circle) can been seen in both implementations as well.

Timing and Scaling Results

To demonstrate the effectiveness of the closest point sparse octree, the construction of the CPSO and the application of UFLIC are timed using varying grid sizes. The amount of time it takes to construct the CPSO scales with the number of sparse voxels. Further, the amount of memory used is significantly reduced in comparison to a full grid.

Three datasets were used for timing purposes: the two datasets used for visual verification , the ICE train and the F6 plane, were voxelized into grids ranging from 512^3 to 4096^3. The third dataset, a vortex coming off a delta wing is also voxelized (Fig. 7), but it is from 512^3 to $8,192^3$.

The timing results, dimensions of the full grid, and the number of sparse voxels of the CPSO are in Table 1. All tests were performed on an Intel Xeon 5170 with 16GB of RAM using a Nvidia Quadro K6000 GPU and CUDA v7.0. The timing results are produced for constructing the closest point sparse octree and running UFLIC. All the UFLIC runs were performed with a life span (ttl) set to 2. Further, UFLIC is run without constructing the neighborhood lookup, for consistent scaling results . All timing results are in seconds.

To save time initializing memory on the GPU, a simple memory pool manager is used. In a preprocess step, a large amount of GPU memory is allocated as a memory pool, which allows for quicker allocation and deallocation of temporary memory buffers when constructing the closest point octree.

For the CPSO construction, the amount of time it takes to construct the sparse octree scales at similar rate as the number of sparse voxels rather than increasing exponentially with the dense grid size. Table 1 includes the timing results for building the CPSO and applying UFLIC to each dataset for varying grid sizes. Further, the number of voxels generated is also in the table.

The ICE train dataset with a grid size of 512^3, 1024^3, 2048^3, and 4096^3 takes 1.12, 3.15, 14.34, and 59.4 seconds to construct the CPSO, respectively. The number of voxels in the sparse octree are 930,803, 3,836,484, 15,800,019, and 65,742,208 for grid sizes 512^3, 1024^3, 2048^3, and 4096^3. For an increase in dimensions from 512^3 to 1024^3, the numbers of voxels increases by 4.1x, and the amount of time to build the CPSO increases by 2.8x. For an increase in the grid size from 1024^3 to 2048^3, the number of voxels increases by 4.1x, and the time to construct the CPSO increases by a factor of 4.6x. Changing the grid size from 2048^3 to 4096^3 increases the voxel count by 4.2x, and the build time for the CPSO increases by 4.1. For each increase in the grid size, both the CPSO and the number of voxels increase linearly at

Table 1: The timing results (in seconds) and the increase in time from the previous grid size for the construction of the CPSO and applying UFLIC as well as dimensions for the datasets are listed. Further, the number of sparse voxels and the increase from the previous grid size voxel count are listed in the last two columns. All timing results were performed with an Intel Xeon 5170 with an Nvidia Quadro K6000 GPU.

	Timing				Dimensions	Sparse Voxels		Sparseness (%)
	CPSO		UFLIC					
	Build (s)	Increase of time	Run (s)	Increase of time		Count	Increase of size	
ICE Train	1.12	-	0.53	-	512^3	930,803	-	99.3
	3.15	2.8x	1.88	3.5x	1024^3	3,836,484	4.1x	99.6
	14.34	4.6x	11.48	6.1x	2048^3	15,800,019	4.1x	99.8
	59.4	4.1	147.84	12.9x	4096^3	65,742,208	4.2x	99.9
F6 Plane	3.22	-	0.31	-	512^3	611,854	-	99.5
	4.5	1.4x	1.42	4.6x	1024^3	2,647,537	4.3x	99.8
	12.85	2.9x	7.63	5.4x	2048^3	11,078,157	4.2x	99.9
	48.84	3.8x	84.31	11.0x	4096^3	45,526,355	4.1x	99.9
edelta vortex	1.71	-	0.06	-	512^3	114,215	-	99.9
	2.25	1.3x	0.29	4.8x	1024^3	489,710	4.3x	99.95
	5.58	2.5x	1.66	5.7x	2048^3	2,594,110	5.3x	99.97
	5.58	2.5x	14.0	8.7x	4096^3	15,259,859	5.9x	99.98
	20.65	3.7x	314.8	19.7x	8192^3	87,677,518	5.8x	99.98

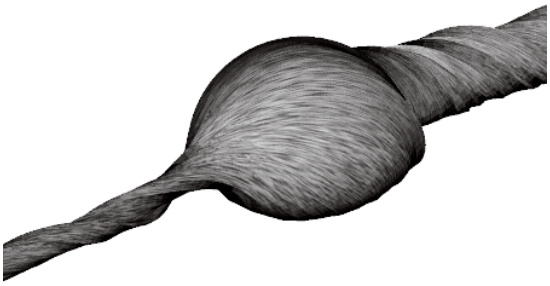


Figure 7: This is the CPSO delta wing vortex bubble with a grid size of $8,192^3$.The delta wing vortex bubble dataset is a stream surface off of a delta wing. It is a complex surface that flows around itself.

a similar rate.

On the other hand, the increase in the UFLIC runtime does not have a linear increase. The increase in time for grid size 512^3 to 1024^3 is 3.5x, the increase in time for grid size 2048^3 is 6.1x, and the increase in time for grid size 4096^3 is 12.9x. The nonlinear increase for the UFLIC time is because the neighborhood index is not used. For instance, without the neighborhood index, applying the Laplacian operator requires eight lookups starting from the root node of the octree tree. This tree traversal is the cause for the UFLIC performance not scaling linearly with the number of voxels.

Table 2 has the ICE Train UFLIC timing results (in seconds) while using the neighborhood index. Increasing the grid size from 512^3 to 1024^3, the time to run UFLIC increased by 0.89 and the number of voxels increased by 4.1x. Similarly, increasing the grid size from 1024^3 to 2048^3 increases the amount of time to apply UFLIC by 4.1x, and the number of voxels increased by 4.1x. By using the neighborhood index, the time to apply the UFLIC increases at a rate similar to the increase in the number of voxels.

Similar to the linear scaling of the CPSO construction and voxel count of the ICE Train, the F6 plane dataset's CPSO construction time increases at a similar linear rate as the voxel count, which can be seen in Table 1. Likewise, applying UFLIC increases at a nonlinear rate because of the required tree traversal.

Finally, the scaling of the CPSO and voxel count of the delta wing vortex bubble dataset are similar to ICE train and F6 plane datasets. Further, applying UFLIC increases at the expected nonlinear rate because of the required tree traversal. The increase in the time to apply UFLIC is not unexpected because the shape of the delta wing vortex bubble is very long and narrow, and the surface folds closely back onto itself multiple times. This closeness requires a higher resolution for the delta wing vortex bubble dataset than the ICE Train and the F6 aircraft. Further, the nonlinear jump seen increasing the grid size from 2048^3 to 4096^3 in the ICE Train and F6 aircraft datasets occurs at the higher, 8192^3, grid size. The nonlinear increase in UFLIC runtime is the same as the ICE train and F6 plane: the neighborhood lookup is not used.

One measure of memory efficiency for the CPSO is comparing the number of voxels in a dense grid to the number of voxels eliminated in the CPSO. All the datasets, regardless of grid size, achieve a 99% or higher sparseness percentage in Table 1, which means that at least 99% of the dense grid is empty data, and the

Table 2: The timing results (in seconds) and the increase in time from the previous grid size for applying UFLIC as well as dimensions for the ICE Train dataset are listed using the neighborhood index. Further, the number of sparse voxels and the increase from the previous grid size voxel count are listed in the last two columns. All timing results were performed with an Intel Xeon 5170 with an Nvidia Quadro K6000 GPU.

	UFLIC Timing		Dimensions	Sparse Voxels		Sparseness (%)
	Run (s)	Increase of time		Count	Increase of size	
ICE Train	0.21	-	512^3	930,803	-	99.3
	0.89	4.2	1024^3	3,836,484	4.1x	99.6
	3.78	4.3	2048^3	15,800,019	4.1x	99.8

sparse octree removed those empty grid cells to save memory.

Finally, compared to the previous technique, the closest point embedding (CPE) [13], the closest point sparse octree scales beyond the CPE memory-limited 1024^3 grid size on the Nvidia Quadro K6000. The CPE is a two-level grid, with the coarse dense grid constructing the closest point grid whereas the refined subgrid is the visualization grid. Unfortunately, constructing the CPSO is not as fast as the closest point embedding, which can construct a closest point grid, with a grid size of $512 \times 58 \times 69$ in 0.03s for the ICE train dataset compared with 512^3 time for the CPSO of 1.12s. For the F6 plane dataset, a closest point grid with a grid size of $384 \times 192 \times 55$ is constructed in 0.06s compared with the 512^3 time for the CPSO of 3.22s. Although significantly faster than our implementation, the CPE cannot skip empty space for the closest point grid construction, and therefore cannot scale to the resolution required to accurately represent the delta wing vortex bubble surface because memory on the GPU is limited.

Conclusion and Future Works

We have introduced a new method for surface flow visualization using the closest point method. The key idea is that by embedding the closest point to a surface into the surrounding grid and extending the surface into the grid, UFLIC can be performed in 3D to generate the 2D embedded surface flow visualization.

Further, we have introduced a sparse octree for the closest point method. Constructing a sparse octree for the closest point method helps save memory over other construction techniques. This expands the ability of the closest point method to larger datasets, which is increasingly important as data sets continue to grow larger over time.

With our new technique, there are numerous advantages compared to previous works. It avoids the visibility problems of image-space approaches, such as popping artifacts on the silhouettes, and can resolve occluded areas that image-space methods cannot. Further, although the user chooses the octree grid size, which is a static constraint, the ability to zoom into intricate areas of the surface which is sometimes needed, can be handled.

In the future, we would like to explore increasing the performance of the octree to bring the runtime down to near interactive rates. Further, we would like to adapt other PDE-based flow visualization techniques such as reaction-diffusion [26].

References

[1] S. Auer, C. Macdonald, M. Treib, J. Schneider, and R. Westermann. Real-time fluid effects on surfaces using the closest point method. *Computer Graphics Forum*, 31(6):1909–1923, 2012.

[2] J. Baert, A. Lagae, and P. Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 27–32, New York, NY, USA, 2013. ACM.

[3] H. Battke, D. Stalling, and H.-C. Hege. Fast line integral convolution for arbitrary surfaces in 3d. In H.-C. Hege and K. Polthier, editors, *Visualization and Mathematics*, pages 181–195. Springer Berlin Heidelberg, 1997.

[4] A. Brambilla, R. Carnecky, R. Peikert, I. Viola, and H. Hauser. Illustrative Flow Visualization: State of the Art, Trends and Challenges. In M.-P. Cani and F. Ganovelli, editors, *Eurographics 2012 - State of the Art Reports*. The Eurographics Association, 2012.

[5] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 263–270, New York, NY, USA, 1993. ACM.

[6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[7] I. Demir and R. Westermann. Vector-to-Closest-Point Octree for Surface Ray-Casting. In D. Bommes, T. Ritschel, and T. Schultz, editors, *Vision, Modeling & Visualization*. The Eurographics Association, 2015.

[8] M. Edmunds, R. S. Laramee, G. Chen, N. Max, E. Zhang, and C. Ware. Surface-based flow visualization. *Computers & Graphics*, 36(8):974 – 990, 2012. Graphics Interaction Virtual Environments and Applications 2012.

[9] L. K. Forssell and S. D. Cohen. Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows. *Visualization and Computer Graphics, IEEE Transactions on*, 1(2):133–141, June 1995.

[10] Y. Hong, D. Zhu, X. Qiu, and Z. Wang. Geometry-based control of fire simulation. *The Visual Computer*, 26(9):1217–1228, 2010.

[11] J. Huang, W. Pei, C. Wen, G. Chen, W. Chen, and H. Bao. Output-coherent image-space lic for surface flow visualization. *2014 IEEE Pacific Visualization Symposium*, 0:137–144, 2012.

[12] G. K. Karch, F. Sadlo, D. Weiskopf, C.-D. Munz, and T. Ertl. Visualization of advection-diffusion in unsteady fluid flow.

Computer Graphics Forum, 31(3pt2):1105–1114, 2012.

[13] M. Kim and C. Hansen. Surface flow visualization using the closest point embedding. *2015 IEEE Pacific Visualization Symposium*, April 2015.

[14] S. Laine. A topological approach to voxelization. In *Proceedings of the Eurographics Symposium on Rendering*, EGSR '13, pages 77–86, Aire-la-Ville, Switzerland, Switzerland, 2013. Eurographics Association.

[15] S. Laine and T. Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM.

[16] R. S. Laramee, B. Jobard, and H. Hauser. Image space based visualization of unsteady flow on surfaces. In *IEEE Visualization*, pages 131–138, 2003.

[17] G.-S. Li, X. Tricoche, and C. Hansen. Gpuflic: Interactive and accurate dense visualization of unsteady flows. In *Proceedings of the Eighth Joint Eurographics / IEEE VGTC Conference on Visualization*, EUROVIS'06, pages 29–34, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[18] G.-S. Li, X. Tricoche, and C. Hansen. Physically-based dye advection for flow visualization. In *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*, EuroVis'08, pages 727–734, Chichester, UK, 2008. The Eurographs Association & John Wiley & Sons, Ltd.

[19] G.-S. Li, X. Tricoche, D. Weiskopf, and C. D. Hansen. Flow charts: Visualization of vector fields on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 14(5):1067–1080, 2008.

[20] C. B. Macdonald and S. J. Ruuth. Level set equations on surfaces via the closest point method. *Journal of Scientific Computing*, 35(2-3):219–240, June 2008.

[21] C. B. Macdonald and S. J. Ruuth. The implicit closest point method for the numerical solution of partial differential equations on surfaces. *SIAM Journal on Scientific Computing*, 31(6):4330–4350, Dec. 2009.

[22] T. März and C. B. Macdonald. Calculus on surfaces with general closest point functions. *SIAM Journal on Numerical Analysis*, 50(6):3303–3328, 2012.

[23] M. Pätzold and A. Kolb. Grid-free out-of-core voxelization to sparse voxel octrees on gpu. In *Proceedings of the 7th Conference on High-Performance Graphics*, HPG '15, pages 95–103, New York, NY, USA, 2015. ACM.

[24] J. Revelles, C. Urea, and M. Lastra. An efficient parametric algorithm for octree traversal. In *Journal of WSCG*, pages 212–219, 2000.

[25] S. J. Ruuth and B. Merriman. A simple embedding method for solving partial differential equations on surfaces. *Journal of Computational Physics*, 227(3):1943–1961, 2008.

[26] A. Sanderson, C. Johnson, and R. Kirby. Display of vector fields using a reaction-diffusion model. In *Visualization, 2004. IEEE*, pages 115–122, Oct 2004.

[27] P. J. Schneider and D. Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[28] M. Schwarz and H.-P. Seidel. Fast parallel surface and solid voxelization on gpus. In *ACM SIGGRAPH Asia 2010 Papers*, SIGGRAPH ASIA '10, pages 179:1–179:10, New York, NY, USA, 2010. ACM.

[29] H.-W. Shen and D. Kao. Uflic: a line integral convolution algorithm for visualizing unsteady flows. In *Visualization '97., Proceedings*, pages 317–322, 1997.

[30] L. Tian, C. Macdonald, and S. Ruuth. Segmentation on surfaces with the closest point method. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 3009–3012, Nov 2009.

[31] J. J. van Wijk. Image based flow visualization. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 745–754, New York, NY, USA, 2002. ACM.

[32] J. J. van Wijk. Image based flow visualization for curved surfaces. In *Visualization, 2003. VIS 2003. IEEE*, pages 123–130, 2003.

[33] D. Weiskopf. Dye advection without the blur: A level-set approach for texture-based visualization of unsteady flow. *Comput. Graph. Forum*, 23(3):479–488, 2004.

[34] D. Weiskopf, G. Erlebacher, and T. Ertl. A texture-based framework for spacetime-coherent visualization of time-dependent vector fields. In *Visualization, 2003. VIS 2003. IEEE*, pages 107–114, 2003.