Surface Flow Visualization using the Closest Point Embedding

Mark Kim* Charles Hansen†

Scientific Computing and Imaging Institute
University of Utah

ABSTRACT

In this paper, we introduce a novel flow visualization technique for arbitrary surfaces. This new technique utilizes the closest point embedding to represent the surface, which allows for accurate particle advection on the surface as well as supports the unsteady flow line integral convolution (UFLIC) technique on the surface. This global approach is faster than previous parameterization techniques and prevents the visual artifacts associated with image-based approaches.

1 INTRODUCTION

Vector field visualization is a fundamental technique in scientific visualization and important in numerous scientific and engineering fields such as computational fluid dynamics. One popular approach is Line Integral Convolution (LIC) [4] because of its efficient utilization of the graphics processor as well as its ability to be used on surfaces embedded in 3D.

Computing LIC on surfaces can be done in two ways: image-space methods and surface parameterization methods. Image-space methods generate LIC images on the visible parts of the surface [12, 27]. In particular, the visible surface geometry and velocity field is projected onto the screen and LIC is applied in the image space. By only processing the visible parts, the computation is highly interactive due to the GPU generated LIC. Unfortunately, there are issues with image-space based methods. Because only the visible geometry is processed, artifacts from altering the camera position can be noticed around silhouette edges or self-occluded areas of the mesh.

Parameterizing the surface is another way to generate LIC on surfaces. Li et al. achieve interactive frame rates rendering unsteady flow by partitioning the mesh into patches which are then packed into a texture atlas [15]. Partitioning the mesh into patches is considered a pre-process step that is very time-consuming.

In this paper we present a new method for unsteady flow line integral convolution on a surface. Our approach is similar to the closest point embedding, a simple technique for solving PDEs on embedded surfaces [20]. By using the closest point embedding, generating the embedded surface can be done at near interactive rates and generating the LIC can be done at interactive rates, allowing flow visualization without the drawbacks of previous methods.

Our contributions are

- Introduce a fast embedded surface for LIC generation that works for arbitrary complex surfaces on the GPU.
- An anti-aliased 3D line algorithm for the closest point embedding.
- An interactive unsteady flow LIC with the closest point embedded surface.

*e-mail: mbk@cs.utah.edu
†e-mail:hansen@cs.utah.edu

To perform the flow visualization, a sparse closest point embedding is constructed by converting the triangular mesh into a sparse three-dimensional closest point grid. Once the sparse closest point embedding is constructed, then a refined grid and a neighborhood index are constructed to visualize the flow. Finally, an unsteady technique, UFLIC, is run over the refined grid to visualize the flow field.

The contents of the rest of the paper are as follows. Related works are in Section 2. Construction of the embedded surface as the closest point grid is covered in Section 3. Constructing the refined grid and adapting UFLIC for the closest point embedding is in Section 4, while the results and conclusions are in Sec. 5 and 6, respectively.

2 RELATED WORKS

2.1 Flow on surfaces

Vector field visualization is a large research field and a comprehensive review is beyond the scope of this paper. Therefore, this review is limited to relevant work on surface-based flow visualization and refers readers to [5] for a more thorough review of flow visualization and surface flow visualization.

Forssell et al. [7] first applied a LIC-based approach to parameterized surfaces by generating the LIC in parameter space. Unfortunately, with this scheme it is difficult to get a distortion free global parameterization. Battke et al. tessellated the surface and performed LIC in the local coordinate space of each triangle [2]. This technique requires a good mesh to perform correctly, limiting its usefulness.

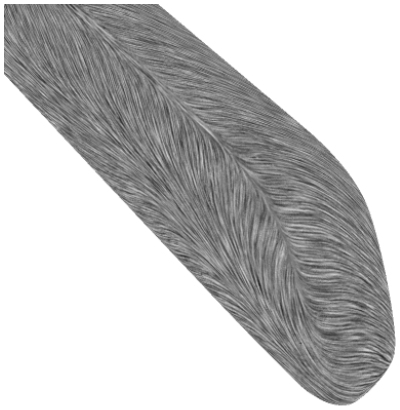
Both the Laramee et al. method called Image Space Advection (ISA) [12] and the Van Wijk method, Image Based Flow Visualization on Surfaces (IBFVS), [27] extend Image Based Flow Visualization [26], or IBFV, a dense texture unsteady 2D flow visualization method, to surfaces. The IBFV method starts with a white noise texture that is warped by the vector field and then blended with other white noise textures over time. Both the ISA and IBFVS extend IBVF by generating, advecting and blending the textures in image space for arbitrary smooth surfaces. Image-based methods are very efficient for arbitrary surfaces with the inherent drawback of artifacts around silhouettes and self-occluding areas. Recently, Huang et al. extended image space based visualization to enhance the coherency of the output [10]. This was done by fixing the triangle-texture matching as well as mipmapping the noise texture. While creating a consistent image, it does not solve the inherent problem of correct surface occlusion nor allow the use of other unsteady flow techniques such as dye advection [14, 11].

Li et al. developed Flow Charts for unsteady flow visualization on surfaces [15]. The Flow Chart method decomposes the triangular mesh into patches with a texture atlas and then the 2D flow is run via a particle system. Once the patches are packed into textures, particle advection schemes for dense texture-based 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 [13, 29, 28]. Finally, this texture is then texture mapped onto the surface during rendering. While Flow Charts is a flexible flow visualization scheme, it has the following drawback: the pre-processing step to decompose the mesh with

a particle system is very time-consuming.

2.2 Closest Point Method

The closest point method (*CPM*) was introduced by Ruuth and Merriman as an embedding surface for solving PDEs [20]. The *CPM*'s usefulness is in its simplicity whereby unmodified $\mathbb{R}^3$ differential operators replace intrinsic surface operators. Macdonald and Ruuth continued the work with an implicit time step, which replaced the original explicit time step as well as evolving a level-set on a surface [17, 16]. Tian et al. followed up the level-set on a surface with segmentation on a surface [25] while Hong et al. apply the *CPM* to the level-set equation to simulate fire on an animated surface [9]. März and Macdonald followed up the works of Macdonald and Ruuth with proofs for the principles of the method [18]. Finally, Auer et al. used the closest point method to solve the Navier-Stokes equations on dynamic surfaces [1].



(a) The Closest Point ICE Train

(b) The Flow Charts ICE Train

Figure 1: The ICE train visualized with UFLIC with the closest point embedding (Fig. 1(a)) and using Flow Charts (Fig. 1(b)).

2.2.1 Closest Point Grid

The closest point method utilizes the closest point grid, which is similar to a discrete distance field [24], except the closest point method is restricted to neither grid points nor facets of a mesh and can represent smooth surfaces. Instead of storing the distance to the surface in the grid, the point on the surface that is nearest to the grid point is stored. This grid is an embedding (the Closest Point Embedding) whereby a surface is represented in the three-dimensional grid.

2.2.2 Equivalence of Gradients

One of the fundamental principles of the closest point method is the "equivalence of gradients" where u is defined as a surface function, $cp(\mathbf{x})$ is the surface point closest to point $\mathbf{x}$ and v is a volume function such that

$$v(\mathbf{x}) = u(cp(\mathbf{x})) \Rightarrow \nabla_s u(\mathbf{x}) = \nabla v(\mathbf{x}) \quad (1)$$

In other words, the gradient on the surface, $\nabla_S u(\mathbf{x})$ agrees with the $\mathbb{R}^3$ gradient of the volume function, v, where v is the closest point extension of u. This makes sense because the closest point extension, $v(\mathbf{x}) = u(cp(\mathbf{x}))$ is constant in the normal direction to the surface, so changes in v must be tangent to the surface.

Further, a second principle concerning surface divergence operators can be derived in a similar fashion to Eq. 1. From these two principles, other differential operators can be constructed, including the Laplace-Beltrami operator [20].

3 Embedding the Surface

The closest point embedding accomplishes two objectives. First, the closest point grid is used to project UFLIC particles back onto the surface (Sec. 3.1). Second, the closest point embedding is used to generate a refined grid and a neighborhood index (Sec. 4.1). This neighborhood index is used to run high pass filtering and anti-aliasing pathlines on the embedded surface at interactive rates. Therefore, the closest point embedding provides a good framework for surface flow visualization.

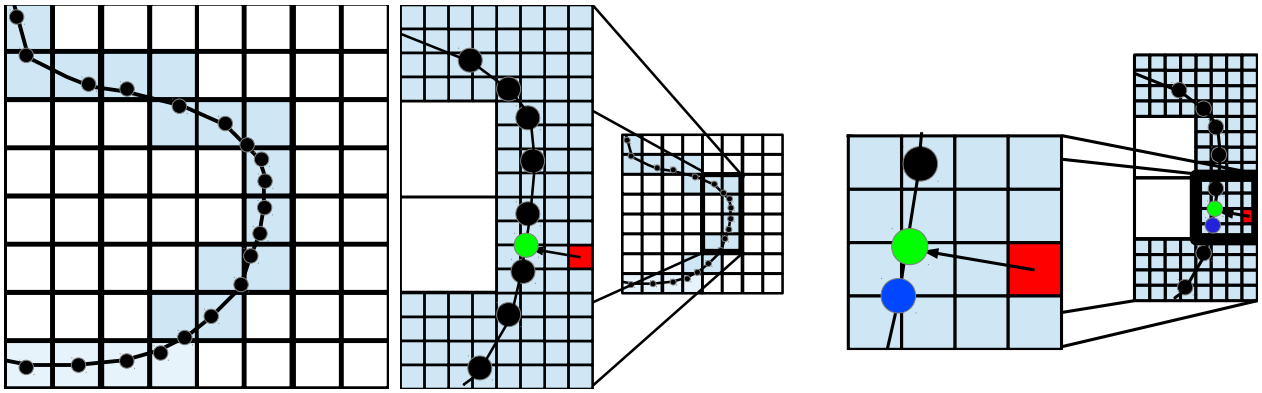
Usually, surface flow datasets are stored as two-dimensional triangular meshes embedded in a three-dimensional space with the velocity field embedded at the vertices of the mesh. To achieve near interactive rates embedding the mesh, Thrust and CUDA are utilized to convert the mesh to the closest point embedding [8, 19]. Constructing the closest point embedding is covered in Sec. 3.1. Once the closest point embedding is constructed, it is used during particle advection to place particles back on the surface, which is covered in Sec. 3.2.

3.1 Constructing the Closest Point Embedding

The closest point embedding is constructed from a surface mesh with the velocity field at the vertices of the mesh. Figure 2(a) is a two-dimensional grid, where the blue cells are close to the surface and the white cells are outside of a narrow band around the surface. The closest point embedding stores the location on the surface that is nearest to the cell. Using Figure 2(b) 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)*.

A two-level grid is constructed to store the closest point embedding, similar to Auer et al [1]. The grid has two levels, a coarse level and a fine level. The coarse level is a three-dimensional grid where each cell represents a block of sub-cells for interpolating the closest point position. The fine level is composed of the sub-cells of the coarse grid cells and is stored in a one-dimensional array. This two-level grid saves memory by only refining the coarse grid where the cells are close to the surface.

Construction of the closest point embedding is in Algorithm 1. The vertices of the surface mesh are binned in the three-dimensional coarse grid. Every cell that contains at least one vertex is marked as "on surface." Figure 2(a) is an example of a one-dimensional curve



(a) The coarse grid. (b) The fine grid. (c) The closest point.

Figure 2: Figures 2(a)-2(c) are two-dimensional examples of the closest point embedding. For all figures, the cells close to the surface are colored blue, while cells far away from the surface are colored white. Figure 2(a) is an example surface, a curve embedded in a coarse grid. Figure 2(b) displays part of the fine level of the surface from 2(a), with spacing $S = 1/4$. An example of the closest point to the surface is shown, where the red cell is at the fine grid position, *(23,14)* and the projection is visualized with an arrow, and the surface location (the green point) is at *(21.3,14.8)*. Finally, Fig. 2(c) focuses on the fine grid cell (from Fig. 2(b)), which is colored red. To determine the closest point on the surface, the surface vertex (in blue) is fetched. Then, the lines adjacent to the vertex are checked to see if there is a point on them closer to the fine grid cell than the surface vertex. In this example, there is a point (colored green) on a line adjacent to the surface vertex that is closer than the surface vertex. The point on the adjacent line is saved to the fine grid.

embedded into a coarse two-dimensional grid. The coarse grid cells colored blue are "on surface" while white cells are considered far away. Next, for each coarse grid cell that is "on surface," it is subdivided to create the fine grid cells. Once all the fine grid cells are determined, then the closest point on the triangular surface is computed and stored in a one-dimensional fine grid array; one for each coarse grid cell.

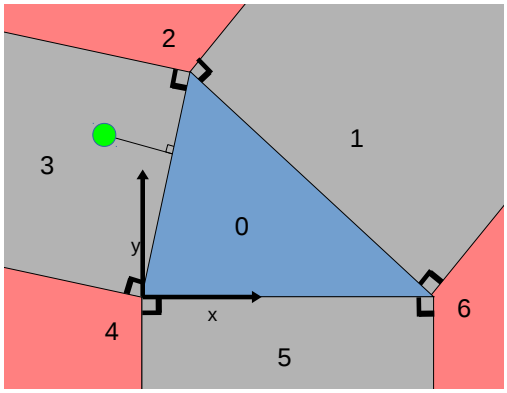


Figure 3: An example of a triangle face (in blue) projected into a coordinate plane and the seven different regions numbered. The green vertex is a grid vertex projected into the two-dimensional plane and is in region 3.

To construct the fine grid cells, each coarse cell that is "on surface" is subdivided into fine cells. Figure 2(b) is a two-dimensional example of six coarse cells (colored in blue), each subdivided into 16 fine grid cells, which are stored in a one-dimensional array. For three-dimensions, the number of subdivisions is 64. For each cell in the fine grid, the vertex on the surface mesh that is nearest to the cell is saved as the current closest point. For each face adjacent to the vertex on the surface, the point on the face that is closest to the grid cell is computed. A two-dimensional example is given in Fig. 2(c). To determine the closest point on the surface, the surface vertex nearest to the fine grid cell is fetched (colored blue). Then, the lines adjacent to the surface vertex are checked to see if there is a point closer to them than the surface vertex. In this example, there is a point (colored green) on the line adjacent to the vertex that is closer to the grid cell than the surface vertex. Therefore, the green point is saved to the fine grid.

In three-dimensions, the adjacent faces to a surface vertex are triangles. To compute the point on a triangle closest to the fine grid cell, the triangle is translated and rotated such that one vertex is at the origin while the two other vertices are in a coordinate plane. This 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 [22]. Figure 3 is an example of a triangle projected into two-dimensions with the seven regions (labeled $0-6$) and a grid vertex, which is in region 3, projected onto the coordinate plane. If this new point on the face is nearer to the fine grid cell than the current closest point, then the current closest point is updated to this new point. This continues until all faces have been processed, and then the closest point is stored in the refined grid cell. The velocity grid is constructed in a similar manner, except the velocity is stored in the grid cell instead of the closest point.

3.2 Using the Closest Point Embedding

Once the triangular mesh is converted to a closest point embedding, a new reprojection step is required to place particles back on the surface after the advection method. To place a particle back onto the surface with closest point embedding, a WENO4 interpolant (Alg. 2) is used to interpolate the position on the surface [6]. For every particle, $\mathbf{p}_i$ the closest point is retrieved from the closest point embedding data structure based on the position of the particle, in one-dimension. This process is repeated for the three cells surrounding the particle because the WENO4 interpolant requires three neighbors for the parabolic interpolation. These are interpolated to compute the location on the surface, $\mathbf{cp}_i$. The particle, $\mathbf{p}_i$ is placed at the location of the interpolated result, $\mathbf{cp}_i$.

Algorithm 1 BuildClosestPointGrid() Input: Triangular Mesh, TM with velocity field VM Output: coarse grid CG, fine grid FG

for all Vertices v_i in mesh TM **do**
  $idx \leftarrow index(v_i)$ ▷ Mark cells in coarse grid as "on surface"
  $CG[idx] \leftarrow True$
end for
for all Cells $cell \in CG$ that are True **do**
  ▷ For all cells that are "on surface"
  for all Fine Grid $FG \in cell$ **do** ▷ Generate subcells
    ▷ Compute the closest point on the surface, cp
    $vtx \leftarrow TM$ vertex nearest to FG
    closest point $cp \leftarrow vtx$
    distance $d \leftarrow \|cp - FG\|$
    ▷ Calculate closest point on faces adjacent to vertex vtx
    for all Face f adjacent to vtx **do**
      $fpt \leftarrow triToEmbedded(f, FG)$
      ▷ *triToEmbedded* returns the point on face f closest to FG (Schneider et al. [22])
      $d_{new} = \|fpt - FG\|$
      if $d_{new} < d$ **then**
        $d \leftarrow d_{new}$
        $cp \leftarrow fpt$
      end if
    end for
    $FG \leftarrow cp$ ▷ Store closest point in grid
  end for
end for

Algorithm 2 $WENO1d(f_1, f_2, f_3, f_4, x)$

$wp_1 \leftarrow parabola(f_1, f_2, f_3, x)$
  ▷ parabola function in Alg. 3
$wp_2 \leftarrow parabola(f_4, f_3, f_2, 1-x)$
$f \leftarrow (wp_1.x \cdot wp_1.y + wp_2.x \cdot wp_2.y)/(wp_1.x + wp_2.x)$
return f

4 Flow Visualization with the Closest Point Embedding

To demonstrate the effectiveness of the closest point embedding for flow visualization, we adapt the unsteady flow line integral convolution, or UFLIC, to visualize surface flow. In this section, we describe constructing the three-dimensional data structure, called the sparsely-stored refined grid, that is used to visualize the flow and adapting UFLIC to the closest point embedding.

Unsteady Flow Line Integral Convolution (UFLIC) is a technique to visualize two-dimensional unsteady flow [23]. 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 is completed, the accumulated values are normalized, filtered and jittered, creating the flow visualization.

Algorithm 3 $parabola(f_1, f_2, f_3, x)$

$F_x \leftarrow (f_3 - f_1) \cdot 0.5$ ▷ first derivative
$F_{xx} \leftarrow f_1 - 2 * f_2 + f_3$ ▷ second derivative
$IS \leftarrow F_x * (F_x + F_{xx}) + 4/3 * F_{xx} * F_{xx}$ ▷ smoothness IS
$IS \leftarrow IS + \varepsilon$ ▷ $\varepsilon = 0.000001$
$IS \leftarrow IS \cdot IS$
$wp.x = (2 - x)/IS$ ▷ weight
$wp.y = f_2 + x \cdot (F_x + 0.5 \cdot x \cdot F_{xx})$ ▷ value at x
return wp

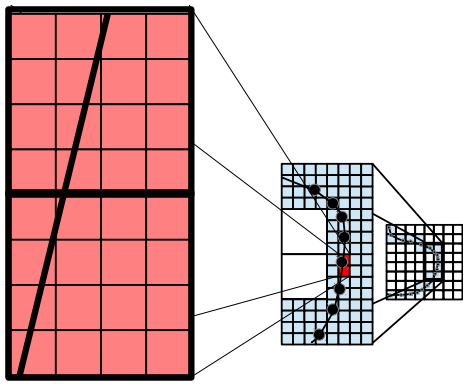


Figure 4: To construct the sparsely-stored refined grid, the closest point embedding is subdivided. Using the original two-dimensional closest point embedding example from Fig. 2, the fine grid is subdivided and two grid cells are each subdivided into eight refined grid cells, in red.

4.1 Construction

To visualize pathlines on the surface, a high resolution data structure, the sparsely-stored refined grid, is constructed. Using the closest point grid size as the refined grid size could result in surface aliasing because it might be too coarse. Globally refining the closest point embedding size would lead to an unacceptable increase in memory. Therefore, the refined grid size is decoupled from the closest point grid size. The closest point grid from Sec. 3 is used to build the refined grid. Once the refined grid is built, a neighborhood index is constructed to speed-up high pass filtering and anti-aliasing the three-dimensional pathlines.

To construct the sparsely-stored refined grid, the closest point grid from Sec. 3 is utilized. The closest point grid is subdivided to refine the grid to suitable levels to visualize the surface. For each cell in the closest point grid that is near the surface, the closest point cell is subdivided into refined grid cells, according to a user-defined parameter, in each dimension. For example, in Figure 4, two cells in the closest point grid (the blue grid) are each subdivided into eight refined grid cells that are highlighted in red.

Once the refined grid is created, the neighborhood index is constructed to speed-up applying the high pass filter and anti-aliasing the pathline because interpolating the closest point for every neighbor lookup is computationally expensive. To construct the neighborhood index, for each refined grid cell, the closest point of the neighboring refined grid cells, ncp_i is computed using the closest point grid and a WENO4 interpolant (Sec. 3.2). Then, the index of the ncp_i is computed, idx_{ncp} and stored in the neighboring index array. By storing the neighboring indices, the Laplacian filter can be applied directly on the refined grid and the anti-aliasing of the pathline is sped-up.

For example, in Figure 5, the green cell is the current cell with an index of cc. The yellow cells are its neighboring cells with indices of rc, lc, uc and dc. In three-dimensions, the neighbor cells would be the neighbors in two-dimensions plus the near and far cells, nc and fc. The neighborhood index for the green cell is $[cc, cc, uc, dc]$ because the right and left neighbors project back into the original green cell.

4.2 UFLIC

To adapt UFLIC to the embedded three-dimensional surface, a piecewise pathline is constructed by advecting the seed particles in three-dimensions, and depositing values onto the surface refined grid. A piecewise pathline is used because the velocity field may

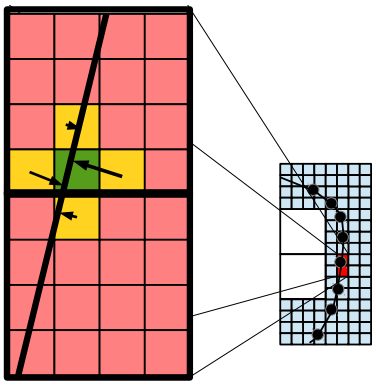


Figure 5: Continuing with the two-dimensional fine grid example from Fig. 4, a single refined grid cell is highlighted in green, with its four neighbors (in two-dimensions) colored yellow.

advect the particle off the surface. If the advected particle is not near the surface, then the pathline is iteratively bisected. This binary search continues until the advected particle is in a grid cell that contains surface. Then the advected particle is projected onto the surface and a line is drawn on the refined grid from the starting point to the advected point. This process is repeated until the length of the piecewise pathline is the same length as the original pathline.

An example is given in Figure 7. In Figure 7(a), the pathline ends off the surface, i.e. in a white cell. The pathline length is cut in half (Fig. 7(b)), but again the pathline terminates off the surface in a white cell. The pathline is halved a third time (Fig. 7(c)) and this time the pathline ends in a blue cell, which contains the surface. A pathline is drawn between the beginning point and the end point, and the end point is projected onto the surface (Sec. 3.2) and becomes the new starting point, as in Figure 7(d).

To draw the piecewise pathline, a three-dimensional Bresenham algorithm [3] is used and adapted for anti-aliasing. To anti-alias the line, a low-pass Gaussian filter is applied to the neighbors in the plane orthogonal to the primary direction of the line. For each grid step, if the step is in the z-axis, the low-pass filter is applied to the xy-plane. Otherwise, if the step is in the y or x-axis, then the xz or yz-plane are updated in a similar fashion, respectively.

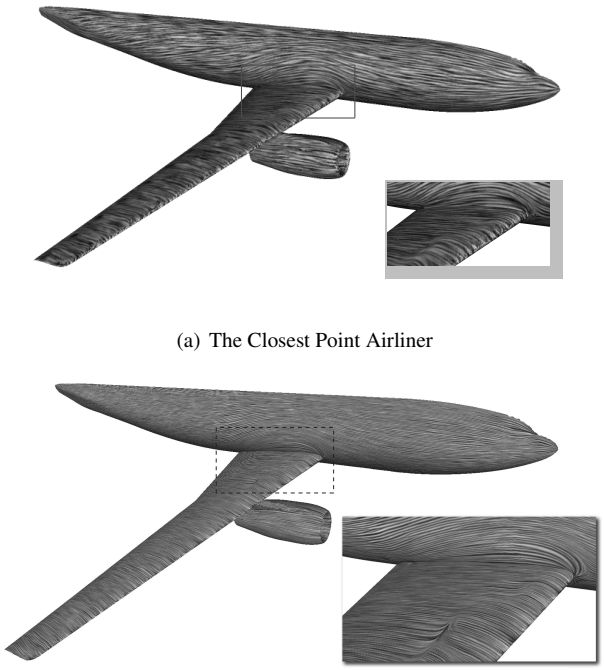
4.3 UFLIC with the Closest Point Embedding

To run UFLIC on the closest point embedding, initially a white noise refined grid is created. Given closest point and velocity grids, the refined grid is constructed as in Sec. 4.1. Once the refined grid is constructed, each refined grid cell is seeded with a particle, and the particle is projected onto the surface using the WENO4 from Sec. 3.2. The particles fetch the velocity from the velocity grid using a linear interpolant and the noise values from the noise refined grid. The particles draw pathlines on the surface as described in Sec. 4.2.

Once all the particles have generated pathlines on the refined grid, a sharpening filter is applied because of the diffusive nature of the UFLIC method [23]. A 3D Laplacian filter is applied to the embedded refined grid by looking up the closest point neighborhood index and fetching the value from the surface cells. Once the filtering is completed, the surface is jittered by adding random values back onto the refined grid and the method is ready for the next iteration.

5 Results and Discussion

To test this new method, three datasets are used: the ICE train, the F6 plane and the cylinder combustion datasets (Figs. 1, 6 and 8



(a) The Closest Point Airliner

(b) The Flow Charts Airliner

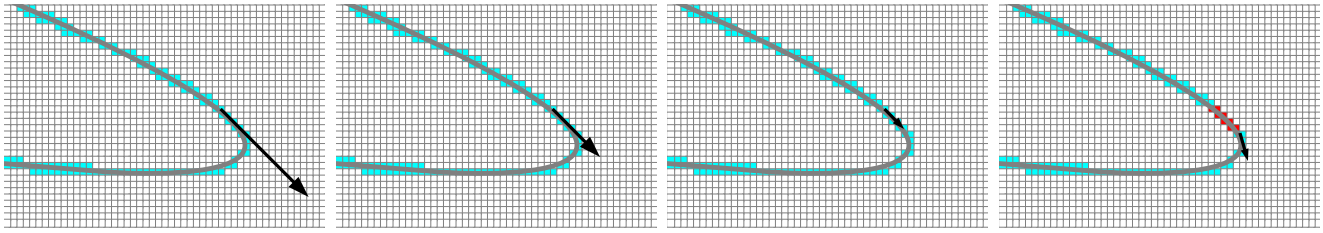
Figure 6: The airliner (F6) dataset visualized with UFLIC and the closest point embedding (Fig. 6(a)) and using Flow Charts in Fig. 6(b).

respectively). An important goal is that the closest point embedding has comparable results to Flow Charts [15], so each dataset has a figure using Flow Charts for comparison purposes.

The ICE train (Fig. 1) 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 attachment flow patterns, which can be seen on the surface in Figure 1(a). Shear stress is shown on the airliner (F6) dataset, which is in Figure 6(a). The combustion dataset (Fig. 8) is a complex combustion cylinder with input and exhaust pipes as well as valves inside the combustion chamber. The swirling flow visualization is aligned with an axis through the cylinder, which is to be expected and can be seen on the cylinder exterior in Figure 8(a).

The timing results and the dimensions of the closest point grid and refined grid for the datasets are in Table 1 and were performed with an Intel Core i7-3770 using a Nvidia GeForce GTX-780 GPU and CUDA v5.5. All tests were performed with a life span (ttl) set to 2. The timing results are produced for constructing the closest point grid, constructing the refined grid and neighborhood index and running UFLIC. All timing results are in seconds. All datasets were constructed and run with less than 1GB of GPU RAM.

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: all the datasets run a maximum of 975,175KB of RAM. The memory is split into two types, temporary and permanent. Permanent data, such as the closest point grid or the grey scale refined grid, are data structures that will last the full iteration. Temporary data is usually helper arrays to compact other arrays in Thrust. Permanent data is added at the head of the memory pool while temporary data is added to the tail of the memory pool. This way, permanent arrays are not interleaved with temporary arrays and the temporary data can be pushed and popped



(a) Original pathline (b) A new bisected pathline from the original pathline 7(a), cut in half. (c) A subline from 7(b), cut in half. (d) A new sub-pathline is started.

Figure 7: Figures 7(a)-7(d) are two-dimensional examples of the pathlines being halved until it is on the surface. In Fig. 7(a), the original pathline does not end in a cell near the surface (cells colored blue). Therefore, the length is cut in half, but again the pathline does not end in a cell near the surface, and the pathline is reduced again (Fig. 7(c)). The pathline now terminates on a cell close to the surface, and a pathline is drawn, shown in red in Figure 7(d). A new pathline is started (in Fig. 7(d)) where the previous pathline ended and using the previous pathline's length. Drawing pathlines in this manner is repeated until the original pathline length is drawn.

Table 1: The timing results (in seconds) and dimensions for the datasets. All timing results were performed with an Intel Core i7-3770 with an Nvidia GeForce GTX-780 GPU.

	Timing (seconds)			Dimensions ($w \times h \times d$)	
	Build CPM	Build Refined Grid	UFLIC	Closest Point	Refined Grid
Ice Train	0.03	0.02	0.1	$(512 \times 58 \times 69)$	$(2048 \times 232 \times 276)$
F6	0.06	0.11	0.12	$(384 \times 191 \times 55)$	$(1536 \times 764 \times 220)$
Cylinder	0.07	0.21	0.17	$(144 \times 222 \times 472)$	$(432 \times 666 \times 1416)$

of the tail of the memory pool without affecting the permanent data. Allocating 975MB as a preprocess takes 0.30s. Allocating on the fly can more than double the runtime, making interactivity difficult.

These experimental results demonstrate a near interactive rate for constructing the closest point grid and an interactive rate for running the UFLIC. The results also show reasonable memory usage with less than 1GB of GPU RAM used for any of the datasets. The timing results for the closest point embedding with UFLIC are similar to the performance of UFLIC with Flow Charts using high resolution textures and a ttl of 2, although it was generated on older GPU hardware.

6 Conclusion and Future Works

We have introduced a new method for surface flow visualization using the closest point embedding. This new scheme achieves interactive rates for performing unsteady flow visualization and a near interactive rate for creating the embedded surface grid. The key idea is that by embedding the closest point to a surface into the surrounding grid, particles can be kept on the surface. Further, the closest point embedding can also perform the high pass filtering required for UFLIC.

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, it does not require a texture atlas like Flow Charts. However, for this implementation a static resolution was chosen, a constraint shared with Flow Charts. This limits the ability to zoom into intricate areas of the surface which is sometimes needed, and can be handled with image-space approaches. This is something we would like to revisit in the future.

In the future, we would like to explore the fast embedding technique to visualize unsteady flow on moving surfaces. Further, we would like to adapt other flow visualization techniques such as UFAC [29] or reaction-diffusion [21] and optimize the amount of memory used. Although a two-level grid is used to save memory, a hierarchical approach would be beneficial to reduce the memory usage and increase the performance.

Acknowledgements

This research was supported by the DOE, NNSA, Award DE-NA0002375: (PSAAP) Carbon-Capture Multidisciplinary Simulation Center, the DOE SciDAC Institute of Scalable Data Management Analysis and Visualization DOE DE-SC0007446, NSF ACI-1339881, and NSF IIS-1162013.

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] 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.

[3] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

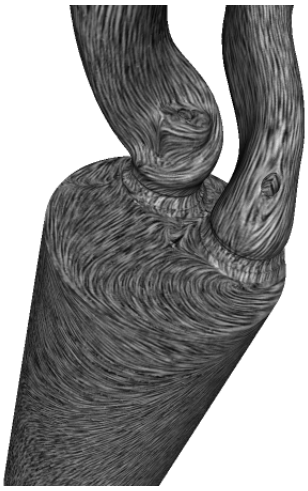
[4] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, pages 263–270, New York, NY, USA, 1993. ACM.

[5] 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.

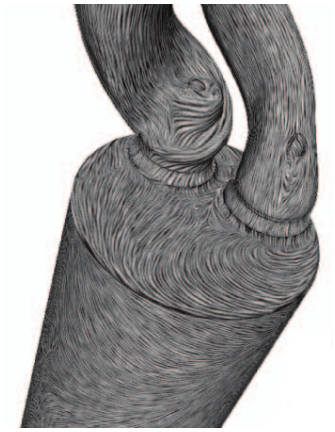
[6] E. Edwards and R. Bridson. A high-order accurate particle-in-cell method. *International Journal for Numerical Methods in Engineering*, 90(9):1073–1088, 2012.

[7] 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.

[8] J. Hoberock and N. Bell. Thrust: A parallel template library. *Thrust: A Parallel Template Library*, 2009.



(a) Closest Point Cylinder



(b) Flow Charts Cylinder

Figure 8: Engine cylinder visualizations. Figures 8(a) and 8(b) use UFLIC with the closest point embedding and Flow Charts, respectively, for visualizing flow in a combustion cylinder.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] G.-S. Li, X. Tricoche, and C. Hansen. Physically-based dye advection for flow visualization. *Computer Graphics Forum*, 27(3):727–734, 2008.

[15] 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.

[16] 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.

[17] 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.

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

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008.

[20] 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.

[21] 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.

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

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

[24] R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of the Sixth Joint Eurographics - IEEE TCVG Conference on Visualization*, VISSYM'04, pages 221–230, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.