GPU-ENABLED SURFACE VISUALIZATION

by

Mark Kim

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2016

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Mark Kim**

has been approved by the following supervisory committee members:

Charles Hansen	, Chair	**11-30-2015** Date Approved
Christopher R. Johnson	, Member	**11-30-2015** Date Approved
Robert Michael Kirby	, Member	**11-30-2015** Date Approved
Ross Whitaker	, Member	**11-30-2015** Date Approved
Guoning Chen	, Member	**12-07-2015** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Visualizing surfaces is a fundamental technique in computer science and is frequently used across a wide range of fields such as computer graphics, biology, engineering, and scientific visualization. In many cases, visualizing an interface between boundaries can provide meaningful analysis or simplification of complex data. Some examples include physical simulation for animation, multimaterial mesh extraction in biophysiology, flow on airfoils in aeronautics, and integral surfaces. However, the quest for high-quality visualization, coupled with increasingly complex data, comes with a high computational cost. Therefore, new techniques are needed to solve surface visualization problems within a reasonable amount of time while also providing sophisticated visuals that are meaningful to scientists and engineers.

In this dissertation, novel techniques are presented to facilitate surface visualization. First, a particle system for mesh extraction is parallelized on the graphics processing unit (GPU) with a red-black update scheme to achieve an order of magnitude speed-up over a central processing unit (CPU) implementation. Next, extending the red-black technique to multiple materials showed inefficiencies on the GPU. Therefore, we borrow the underlying data structure from the closest point method, the closest point embedding, and the particle system solver is switched to hierarchical octree-based approach on the GPU. Third, to demonstrate that the closest point embedding is a fast, flexible data structure for surface particles, it is adapted to unsteady surface flow visualization at near-interactive speeds. Finally, the closest point embedding is a three-dimensional dense structure that does not scale well. Therefore, we introduce a closest point sparse octree that allows the closest point embedding to scale to higher resolution. Further, we demonstrate unsteady line integral convolution using the closest point method.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

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

CHAPTER 1

INTRODUCTION

Surface visualization is a fundamental technique in scientific visualization that facilitates understanding of surface data. It covers a wide range of data types and techniques, from mathematical algebraic surfaces to raster data and volume rendering to mesh extraction. Regardless of data type, though, there is usually a need for sophisticated but fast techniques. Further, surface visualization can be used as part of an iterative process to explore data sets. This iterative process requires a reasonable turnaround time for generating results because a lengthy turnaround time inhibits exploration of the data set. As sophisticated surface visualization techniques are used to increase the quality of the output, in contexts such as medical imaging [94] or surface flow visualization [52], these techniques come with an additional computational cost. That sophistication can also increase the amount of time it takes to iterate scientific discovery in a timely but accurate manner [107]. But this sophistication is key to accurate understanding of the data and cannot be discarded for performance. To address these issues, we developed new parallel techniques for surface visualization.

Surface visualization aids in the understanding of surface data, and in this dissertation we focus on two areas: conformal meshing with particles and unsteady surface flow visualization. Bioengineers need conformal, multimaterial meshes for accurate simulation [108]. In contrast to meshing, unsteady flow visualization, such as UFLIC, help engineers understand the flow on surfaces over time. Although with very different purposes, both of these research areas have similar needs for exploring their data in a timely manner while maintaining the accuracy of the results. Additionally, surface visualization covers a broad spectrum of data types and techniques. Three-dimensional structured scalar data, such as those generated from magnetic resonance imaging (MRI) or computed tomography (CT), are frequently used in biomedical research for electrophysiological modeling. To conduct the modeling, a tetrahedral mesh is extracted from the structured data using a software package called BioMesh3D [98]. To perform accurate electrophysiological modeling, the mesh needs to be a conformal, high-quality multimaterial tetrahedral mesh [108]. An example of a five-compartment mesh used in electrophysiology is in Fig. 1.1. In contrast to a structured, three-dimensional scalar field, surface flow visualization often uses a triangular mesh with a velocity field sampled at the mesh vertices. With a velocity field represented in the mesh, various surface flow techniques are applied to visualize the field, such as line integral convolution (LIC), which has a physical analog: placing oil onto aircraft to visualize flow separation (Fig. 1.2). Although dissimilar in structure and purpose, both multimaterial meshing and the surface flow visualization share a similar problem: sophisticated visualization techniques have a high computational cost for accurate visualization.

Surface visualization, and scientific visualization in general, is often used as part of an iterative data exploration process. This iterative data exploration process is a process of trial and error to explore the data. For example, in BioMesh3D there are user parameters to control some aspects of the mesh generation. If the turnaround time is lengthy, in practice this can lead to users resisting a technique or being unable to fully explore the data to generate an optimal solution. Therefore, it is preferable that surface visualization techniques be fast enough for users to iterate

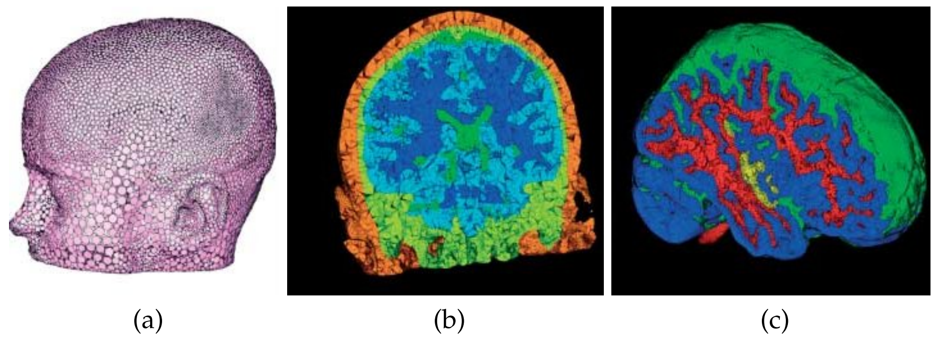


Fig. 1.1. Five-compartment tetrahedral mesh using BioMesh3D. Fig. (a) visualizes the particles on the surfaces. Fig. (b) and (c) visualize the tetrahedral mesh. (MacLeod *et al.* [73])

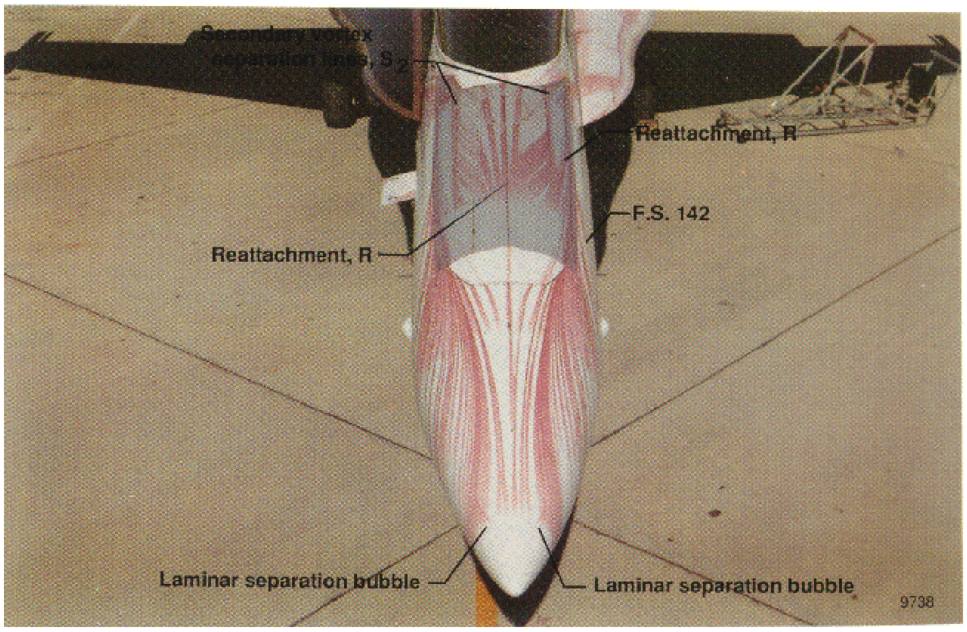


Fig. 1.2. In-flight oil flow [33] (NASA [33]).

over their parameter space in a timely manner. At the same time, the high-quality results from the sophisticated techniques need to be maintained.

Users such as bioengineers or computational fluid dynamicists require increasingly sophisticated surface visualization for more accurate results, and this requirement increases computation time. The type of conformal multimaterial meshes used for biomedical electrophysiological modeling comes with a high computational cost, on the order of hours, or even days, to generate, that inhibits the ability of biomedical engineers to generate many models over time [107]. Similarly, state of the art in parameterized surface flow visualization, Flow Charts, requires a lengthy preprocess step to generate a parameterized surface upon which to perform unsteady flow line integral convolution [65]. These lengthy times to generate results can interfere with the iterative nature of using surface visualization as an exploratory tool for data sets. But these techniques are crucial to their users for high-quality scientific discovery.

To address these issues, new parallel techniques were developed to be used in combination with general purpose computing on the graphics processing unit (GPGPU) to speed up surface visualization. At the same time, care is taken to retain the sophistication, complexity, and accuracy of the techniques that are required to continue to be useful for practitioners in their field. This chapter continues with an overview of the contributions in this dissertation for surface visualization.

1.1 Particle-Based Mesh Extraction on the GPU

Isosurface extraction from three-dimensional scalar volumes is a fundamental technique in visualization. In some cases, the scalar data may be composed of different materials, and although the material is stored in a regular grid, the material interfaces generally do not conform to the underlying grid. Meyer *et al.* [80, 81] introduced a particle-based approach to extract a conformal, curvature-dependent, well-formed multimaterial mesh from biological data. This approach used an energy-based system to extract a surface mesh with nearly equilateral triangles. Further, it generated meshes with smaller triangles in areas of high curvature, which gives more resolution in areas that need it. Well-formed triangular meshes are a good starting point to generate a tetrahedral mesh that is well suited for finite element simulation.

BioMesh3D [98] is a tool based on the research of Meyer *et al.* and packaged into a pipeline for biological mesh extraction [19]. However, due to the computational complexity of the particle advection process, users are required to find a balance between the heavy computation required and their needs in terms of the quality of the mesh, the quantity of tetrahedrons, and the time anticipated to extract the mesh. The excessive computational cost to generate a well-shaped multimaterial mesh has hindered the use of the curvature-dependent particle system by the bioengineering community for numerical simulations [108]. For instance, an attempt was made to extract a mesh from a six-material dataset, but was finally stopped after two months because it had yet to finish [107]. Improving the performance could increase the use of the particle system for multimaterial mesh extraction and for various numerical simulation tasks.

We introduced a novel implementation of a particle system on the graphics processing unit (GPU) to reduce the run-time of the particle system. The adaption of a particle system to the GPU is inherently difficult due to the single instruction multiple threads (SIMT) nature of the hardware. We studied the potential parallelization of the particle placement and proposed a simple strategy, called the red-black update, to segment the particles into groups that can be processed concurrently. Then, we explored the parallel feature provided by the recent advance of CUDA programming on the GPU, which allowed us to parallelize the computations when processing each particle in a group. Finally, we applied our GPU-based particle system to a number of medical datasets. The obtained meshes have comparable quality to those generated using a CPU-based particle placement, whereas the computation of our implementation is at least one order of magnitude faster than the CPU version for most cases, which is described in detail in Chapter 3.

1.2 Enhanced Particle-Based Mesh Extraction

The red-black update scheme achieved up to an order of magnitude speed-up for isosurface extraction using a single distance field on the GPU. Although it performed well on the GPU, unfortunately, it is not a natural mapping to the SIMT architecture. Further, extending the red-black scheme to multiple materials was problematic because of the SIMT nature of the GPU. The surface representation, a distance field, requires a reprojection step realized through an iterative root-finding algorithm to place particles back onto the surface. This reprojection step is inefficient on the GPU due to the amount of control flow, and forced the red-black update to run inefficiently on the GPU. Therefore, a new approach is needed to overcome these issues. We used the closest point embedding to define the surfaces in the volume for faster reprojection. We adapted the Barnes-Hut tree code, an octree-based acceleration structure, to speed up the particle energy calculation. Finally, new seeding and add/delete algorithms were developed to efficiently place new particles.

1.3 Surface Flow Visualization

Vector field visualization is a fundamental technique in scientific visualization and is important in numerous scientific and engineering fields, such as computational fluid dynamics. One popular approach is line integral convolution (LIC) [17] because of its efficient utilization of the graphics processor as well as its ability to be used on surfaces embedded in three dimensions.

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 [61, 111]. In particular, the visible surface geometry and velocity field are projected onto the screen, and LIC is applied in the image space. By processing only the visible parts, the computation is highly interactive due to the GPU-generated LIC. Unfortunately, there are issues with image-space 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.* achieved interactive frame rates rendering unsteady flow by partitioning the mesh into patches that are then packed into a texture atlas [65]. Partitioning the mesh into patches is considered a preprocess step that is very time consuming.

To avoid the artifacts from image-space methods while at the same time addressing the lengthy preprocess step of Flow Charts, we presented a new method for unsteady flow line integral convolution (UFLIC) on a surface. Our parameterized space, the closest point embedding, came from the closest point method a simple but powerful technique for solving PDEs on embedded surfaces [93]. By using the closest point embedding, the parameterized surface was generated at near-interactive rates and the UFLIC was done at interactive rates, which allowed for flow visualization without the drawbacks of previous methods. To perform the flow visualization, a sparse closest point embedding was constructed by converting the triangular mesh into a coarse three-dimensional closest point grid. Once the closest point embedding was constructed, a refined grid and a neighborhood index are constructed to visualize the flow. Finally, an unsteady flow technique, UFLIC, was run over the refined grid to visualize the flow field.

1.4 Surface Flow Visualization and the Closest Point Sparse Octree

Finally, as datasets continue to grow larger, new methods are needed to visualize them. To address this issue, we introduced the closest point sparse octree [53]. By using a sparse octree instead of a structured grid, we could construct a closest point embedding up to $8,192^3$ in size on the GPU. Further, previously the closest point embedding was used to keep particles on the surface to perform UFLIC on the surface. However, by using the closest point method, particles no longer are kept on the surface to advect the noise. By extending the surface velocities and values into the surrounding grid and advecting the particles in the three-dimensional grid, the UFLIC is performed on the surface thanks to the *equivalence of gradients* [93].

1.5 Contributions

This dissertation explores speeding up surface visualization through the use of the GPU with the closest point embedding. The following contributions have been made:

- *Particle System for Meshing on the GPU*. The curvature-dependent particle system [80] is adapted to the GPU for isosurface mesh extraction [50]. A red-black Gauss-Seidel update method was developed to process bins of particles in parallel. This method achieved up to 44x speed-up over a CPU version.

- *Particle Mesh Extraction With the Closest Point Embedding and the GPU*. Unfortunately, the GPU particle system in [50] had some limitations. Although it achieved an order of magnitude speed-up over a CPU implementation, nevertheless it was difficult to implement and extend to multiple materials, and work load balancing was restricted. To remedy this, a tree code is used instead of binning to increase performance by not limiting the acceleration structure by the largest local feature size value. Further, a closest point embedding surface is used instead of multiple distance fields to immediately project the particle back onto the surface. By choosing the closest point embedding, multiple projections onto the surface are no longer needed [51].

- *Surface Flow Visualization Using the Closest Point Embedding*. The closest point embedding is a powerful tool to represent surfaces. Therefore, it is applied to surface flow visualization [52]. Previous surface flow visualization attempts are either at least an order of magnitude too slow to parameterize the surface in near real-time or image-space based and unable to support techniques such as dye advection [63, 61, 111]. The closest point embedding is constructed in near real-time, and UFLIC [99, 63] is adapted to the closest point embedding to do surface flow visualization.

- *Closest Point Sparse Octree and Unsteady Surface Flow*. As datasets continue to grow larger, so must the methods adapt to keep up with the increased size. Therefore, we introduce a GPU-based closest point sparse octree (CPSO) construction technique [53]. This new technique can construct sparse octree grids up to $8,192^3$ in size. Further, we introduce the closest point method to surface flow visualization by using unsteady flow line integral convolution (UFLIC) with the closest point method.

1.6 Outline

The remainder of this dissertation is as follows. Chapter 2 outlines the previous works of particle systems, surface flow visualization, the closest point method, and sparse octree voxelization. For particle systems, the natural focus is on particle systems on the GPU in Section 2.1.1 and the Barnes-Hut tree code (Section 2.1.2). The mesh extraction background reviews variational methods in Section 2.2. Multimaterial mesh extraction methods, besides variational, are briefly covered in Section 2.2.3. For flow visualization in Section 2.3, the focus is on the fundamentals of surface flow visualization and image-space versus parameter-space approaches (Section 2.3.3). The closest point method is covered in Section 2.4, and recent sparse octree voxelization strategies are covered in Section 2.5.

Chapter 3 reviews the work done to adapt the cotangent energy particle system to the GPU using the red-black update. Section 3.1 provides an overview of the serial cotangent energy particle system. Then, we introduce the reasoning behind the red-black update in Section 3.3 and the nuts-and-bolts of the red-black update on the GPU. Finally, in Section 3.6 we discuss the difficulties in the extension of the red-black update to multiple materials.

Chapter 4 describes the enhanced mesh extraction for the GPU. In Section 4.1, we explain why the Barnes-Hut tree code is better suited for the GPU than the red-black update and how it is adapted to the GPU. Then, we discuss the closest point embedding and how it is used for particle mesh extraction in Section 4.2. We then compare the results of this enhanced mesh extraction with our previous red-black update scheme in Section 4.3.

Chapter 5 continues with the closest point embedding, but we adapt it for surface flow visualization. Section 5.1 introduces the closest point embedding and constructs the coarse and refined grid from a triangular mesh and reprojects particles back onto the surface. Then, we present UFLIC and how it is applied to the closest point embedding in Section 5.2. Finally, we compare previous results with our results in Section 5.3.

Chapter 6 discusses the closest point sparse octree with unsteady surface flow visualization. By using a sparse octree to represent the closest point embedding, the grid can scale up to $8,192^3$ in size. Further, the unsteady flow is accomplished using the closest point method, where the velocities and values on the surface are interpolated off the surface to perform the advection, depositing, and filtering in three dimensions. Finally, Chapter 7 includes the conclusion and future works.

CHAPTER 2

BACKGROUND

This chapter provides the background for subsequent chapters. Particle systems on the graphics processing unit (GPU) are reviewed in Section 2.1, and Section 2.2 discusses mesh extraction with an emphasis on variational methods (Section 2.2.2). These sections are relevant to Chapters 3 and 4. Flow visualization is reviewed in Section 2.3 with a focus on surface flow visualization (Section 2.3.3), which is relevant to unsteady flow line integral convolution on a surface in Chapters 5 and 6. Sparse voxelization is reviewed in Section 2.5, which pertains to the closest point sparse octree in Chapter 6. The closest point method is discussed in Section 2.4, which is revelant to Chapters 4 – 6. Finally, GPU computing is reviewed in Section 2.6.

2.1 Particle Systems

Particle systems are an expansive topic in computer science. In this section, we limit the topics to particle systems for scientific visualization on the GPU (Section 2.1.1) and the Barnes-Hut tree code on the GPU (Section 2.1.2).

2.1.1 Particle Systems on the GPU

Particle systems on the GPU were first introduced by Kolb *et al.* [56] and Kipfer *et al.* [54] for real-time animation and rendering of particles in OpenGL. For real-time three-dimensional flow visualization, Kruger *et al.* used a particle system on the GPU because the CPU was too slow [57]. Extending the particle system beyond computer graphics, the GPU was subsequently used for simulating fluid motion with smooth particle hydrodynamics (SPH) [55]. A good overview of state-of-the-art in SPH on the GPU can be found in Goswami *et al.* [39].

Although there are shared characteristics between these particle systems and the system presented in this dissertation, such as how particles are stored and accessed on the GPU, each has a different parallelization strategy due to different application purposes. The particle systems by Kolb *et al.* [56] and Kruger *et al.* [57] do not require neighborhood information and are easily parallelizable (i.e., assigning a thread for each particle). On the other hand, Kipfer *et al.* [54] and the SPH implementations [39, 55] require local neighbors for collision detection and advection of the particles. However, both of these systems are Forward-Euler solutions, which could use a small uniform time step to adjust the particle velocity to allow the systems to converge. In our implementation, each particle determines its step size based on its energy and the local curvature and does not have a uniform time step, which allows faster convergence for the purpose of mesh extraction.

2.1.2 Barnes-Hut Tree Code

The original tree code by Barnes and Hut was an astrophysics simulation that transformed the N-Body problem from an order $O(n^2)$ problem to $O(nlog(n))$ [4]. Briefly, all particles in the domain are stored in an octree such that each leaf in the octree has either zero or one particle. Each internal node represents its children, the group of particles beneath it, in the tree. To represent children, an internal node stores a center-of-mass location and the total mass of all its children particles.

Following the original hierarchical Barnes-Hut tree code, there were numerous attempts to parallelize the method on vector hardware of the time [5, 28, 112], sometimes with no speed-ups over direct N-body simulations [75]. At the same time, custom hardware was developed to solve the direct N-body problem called “GRAvity PipE” (GRAPE), which provided two orders of magnitude speed-up in comparison to software implementations [74]. GRAPE hardware was modified to compute the gravitational forces for Barnes-Hut [36], and the hardware and associated libraries continued to evolve and be used by researchers in astrophysics.

Because of the highly parallel nature of GRAPE and the addition of SIMD extensions (SSE) instructions on processors, the GRAPE library was implemented for the CPU [84]. The highly parallel nature of the SIMT instructions of the GPU and a parallel programming model, common uniform device architecture (CUDA), also led to the GRAPE hardware API being adapted to the GPU [38]. The advent of programmable GPUs and their significantly higher volumes caused a decline in the use of GRAPE hardware because GPUs are more cost efficient and cheaper to buy [7].

An early GPU implementation of tree code by Belleman *et al.* was faster than the CPU tree code, but slower than direct N-body methods on the GPU [9]. Hamada *et al.* increased the speed-up by combining multiple tree-walks and then transferring them to the GPU [41], and Gaburov *et al.* improved upon this by moving the tree-walk to the GPU [37]. Finally, Bédorf *et al.* [8] implemented the remaining parts of the tree code, constructing the octree and particle sorting, solely on the GPU. A comprehensive overview of parallel Barnes-Hut tree code can be found in [7, 117].

2.2 Variational Mesh Extraction

A comprehensive review of meshing is outside the purview of this dissertation. Therefore, the focus in this section is on variational meshing. For a more comprehensive overview of mesh extraction, we refer readers to Shewchuk [102].

2.2.1 Quality

In two dimensions, variational optimization techniques such as Meyer's cotangent energy system or Centroidal Voronoi Tessellation are effective at generating high-quality isotropic triangular meshes. Variational optimization strategies rely on the idea that well-spaced points lead to well-shaped isotropic triangular meshes [31]. Therefore, isotropic triangulation can be reshaped into a point sampling problem, whereby if the points are distributed evenly on a two-dimensional domain, the resulting mesh has well-shaped triangles. Numerous strategies accomplish this goal. Three major ways to tessellate in two dimensions that attempt to satisfy this "well-spaced" criterion include Quadtree, Delaunay refinement, and disk packing. Quadtrees repeatedly subdivide the domain, but the result can be biased towards horizontal and vertical edges. The Delaunay refinement eventually becomes well spaced. Finally, disk packing is not proven, but "seems straightforward" [31].

Well-spaced particles in two dimensions generate high-quality triangulations, and well-spaced points can be generated and the number of points bounded in any dimension [31]. However, Delaunay triangulation of well-spaced particles in three dimensions does not generate good tetrahedral meshes. Listed below are three methods to measure the quality of a tetrahedron. Minimum sine angle (Freitag and Oliver-Gooch [35]) is the minimum of the six dihedral angles of the faces and is in Fig. 2.1b. Volume length measure (Parthasarathy *et al.* [88]) and radius ratio (Cavendish *et al.* [18]) are similar in that they measure the ratio of volume to a side (Fig. 2.1a).

In three dimensions there are six classifications for tetrahedra: round, needle, wedge, spindle, sliver, and cap (Fig. 2.2). Eppstein notes that ”well-spaced point sets form only round and sliver tetrahedra” [31] and Alliez *et al.* [1] explain it as: ”We can attribute the slivers to the fact that [CVT] tends to optimizes the compactness of the dual Voronoi cells, but not the compactness of simplices in the primal Delaunay triangulation: therefore, the presence of a sliver is not penalized by this energy.” In other words, the CVT optimizes on the (dual) Voronoi side and ignores packing on the (primal) triangulation side. Other measurements can alleviate this problem, however. For instance, a minimum sine does not penalize some tetrahedrons, such

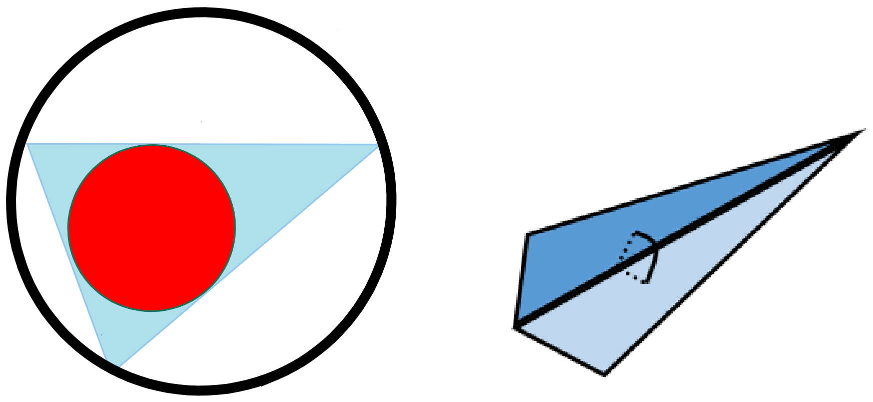

(a) Radius ratio (b) Dihedral angle between two triangles.

Fig. 2.1. Examples of measuring triangle quality. In (a) is the inscribed vs the circumscribed radius ratio of a triangle and (b) is the angle between two triangles.

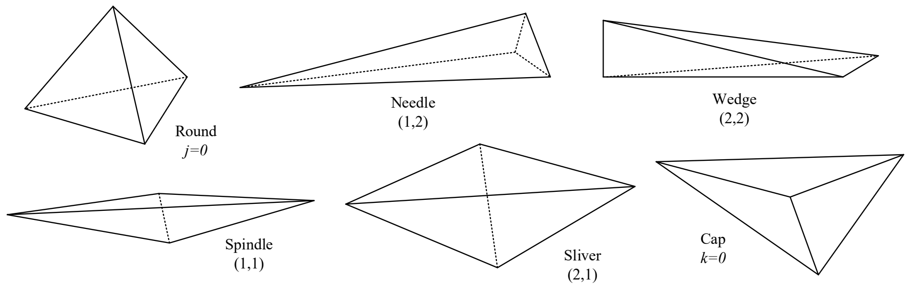


Fig. 2.2. A collection of different tetrahedrons and their descriptions [31]. A round tetrahedron has a good aspect ratio. A needle has good dihedral angles with a small solid angle and a wedge has small dihedrals and solid angles. A spindle has small solid angles and wide dihedrals. A cap tetrahedron has wide solid angles and a sliver is considered bad for computation.

as needles. However, needle tetrahedrons with good dihedral angles might be "harmless" [103].

Therefore, in two-dimensional space, well-spaced points generate well-shaped triangular meshes. In three dimensions, however, well-spaced points do not generate good tetrahedral meshes; rather they form meshes with round and sliver tetrahedrons.

2.2.2 Variational Methods

Some more modern examples of variational methods are briefly discussed below. Du and Emelianenko proposed Centroidal Voronoi Tessellation, which uses Lloyd's relaxation to iteratively move the generating point of a Voronoi cell to its centroid [27]. Unfortunately, Lloyd's relaxation can be slow, but there are methods to speed it up [26, 67]. Witkin and Heckbert were among the first to use particles for visualization [116]. They used an energy-based particle system to visualize implicit functions. They chose to use a Gaussian energy function based upon the distance from a particle to its neighbors to evenly place particles on the surface. The energy of a particle repelled its neighbors, which, after a number of iterations, place particles evenly on the surface. Following the lead of Witkin and Heckbert in the use of particles for visualization, Crossno and Angel used a particle system to extract isosurfaces from scalar fields [22]. Shimada and Gossard developed Bubble Mesh, a nonlinear force system to pack spheres [104].

Meyer *et al.* employed an energy-based particle system for visualizing implicit surfaces [79], and extracting high-quality meshes from scalar fields [80]. Instead of the Gaussian energy function used by Witkin and Heckbert [116], Meyer *et al.* applied a compact cotangent energy function because it is approximately scale invariant. Additionally, Witkin and Heckbert used a gradient descent to minimize the energy, which requires a tuning parameter. Meyer *et al.* replaced the gradient descent with a Gauss-Seidel update and used an inverse Hessian scheme to automatically tune the energy minimization, removing this tuning parameter. Finally, this method allowed for the placement of more particles near areas of high curvature, while leaving regions of low curvature with fewer particles and fewer tuning parameters. Bronson *et al.* introduced a particle-based system for generating adaptive triangular surfaces and tetrahedral meshes for CAD models [15]. Instead of precomputing feature size, their system adapts to curvature and moves the particles in the parameter space.

2.2.3 Other Multimaterial Meshing Techniques

However, particle-based mesh extraction has no guaranteed bounds. A class of Delaunay refinement multimaterial mesh generators (Pons *et al.* [90] and Boltcheva *et al.* [12]) used a sliver-removal technique [20] and demonstrated good results. Instead of capturing the surface and infilling volumes, lattice techniques construct a background structure, for example a three-dimensional octree, and recover the surface by finding intersections between the surface and the mesh. The most widely used example is marching cubes [70]. Zhang *et al.* expanded the octree-based dual contour meshing scheme [118] to multiple materials [119]. Liu *et al.* used a two-step mesh decimation process to extract multiple materials from a background lattice [68]. Bronson used a background lattice in combination with mesh warping to produce bounded-quality meshes [14].

2.3 Flow Visualization

Flow visualization is an expansive topic; therefore, the focus in this section is on modern dye and texture advection and flow over surfaces. For a comprehensive overview of flow visualization, we refer the reader to Laramee *et al.* [60], and for surface flow visualization, Edmunds *et al.* [29].

2.3.1 Dye Advection

In physical experimentation, adding a tracer such as smoke or dye to aid in visualizing fluid flow is a common occurrence. Max *et al.* introduced flow volumes as a smoke tracer and three-dimensional equivalent of streamlines for vector field visualization [78]. Shen *et al.* [100] advected dye in texture space to highlight features in combination with LIC visualization, and Jobard *et al.* [46] used GPU hardware to advect and blend the dye texture. Van Wijk used image-based flow visualization (IBFV) to inject a dye and advect it forward.

Semi-Lagrangian advection is a hybrid Lagrangian-Eulerian method that advects particles from the Eulerian grid points and resamples them in the next time step back to the grid, usually with backwards time integration [23, 48, 64, 114]. Unfortunately, because of this continous interpolation onto an Eulerian grid, semi-Lagrangian methods suffer from numerical diffusion. Jobard *et al.* tackled this with a sharpening function [48], and Weiskopf used a level-set to model the dye interface [114]. The level-set is used to represent the interface between dye and background materials and periodically reinitialized to correct the diffusion. Unfortunately, the level-set still has numerical diffusion, and Cuntz *et al.* attempted to correct this with particles [23]. Li *et al.* used a "control volume" instead of point samples and a piecewise-parabolic interpolant to minimize the numerical diffusion in dye advection [64].

Unfortunately, semi-Lagrangian methods are not mass preserving. Therefore, Karch *et al.* employed a dimension-splitting WENO-based finite volume flux through the faces scheme to decrease the numerical diffusion and enforce conservation [49]. Although this method is mass conserving and interactive, it is now constrained by the Courant-Friedrich-Levy (CFL) condition, whereby a velocity step must be smaller than the grid size, which reduces performance because more steps must be taken. Nonetheless, it is an accurate, mass-conserving, and interactive method.

2.3.2 Line Integral Convolution

For this dense texture advection section, we focus on the vector field visualization family of line integral convolution, which was introduced by Cabral and Leedom [17]. Line integral convolution is a texture advection technique for visualizing vector fields. Briefly, a noise field is convolved along bidirectional streamlines using a low-pass filter. In particular, given a streamline σ and a pixel at position $\boldsymbol{x}$, the intensity I at $\boldsymbol{x}$ is:

$$I(\boldsymbol{x}) = \int_{s_o+L/2}^{s_0+L/2} k(s-s_0)T(\sigma(s))ds \tag{2.1}$$

where s is the arc length to decompose the streamline curve, T is the input noise texture, k is the low-pass filter, and L is the width of the filter.

There have been various attempts to speed up LIC, with the most successful being that by Stalling and Hege [105]. Here, a key observation is made: there is a significant amount of redundant streamlines and kernel convolution work to be reused.

Another way to speed up LIC is to parallelize it. One of the earliest attempts to parallelize LIC was that of Zöckler *et al.* [120], who used a massively parallel "Cray T3D" by taking advantage of temporal coherence between frames. However, it would be a few more years until commodity graphics would come along to allow for parallel LIC without the massive hardware [46, 47, 110].

To visualize time-varying data, Shen and Kao extended line integral convolution and called it unsteady flow line integral convolution or UFLIC [99, 101]. Instead of streamlines and a low-pass filter, UFLIC integrates over pathlines and uses a high-pass filter and noise jittering to generate a smooth, temporally coherent dense texture. Shen and Kao also extended UFLIC to shared-memory multiprocessor computers [101]. Unfortunately, UFLIC can be slow. Attempts have been made to increase the performance, similar to Stalling and Hege, by reusing pathlines and convolutions [69]. Li *et al.* adapted UFLIC to the GPU (GPUFLIC) by using texture hardware for the particle advection as well as the graphics hardware to deposit the values [63]. By adapting UFLIC to the GPU, they were able to achieve interactive rates with GPUFLIC.

2.3.3 Flow on Surfaces

Dye and texture advection has seen success in both two dimensions and three dimensions, but visualization of flow on surfaces is more limited. Forssell and Cohen [34] first applied an 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 [6]. This technique requires a good mesh to perform correctly, limiting its usefulness. Both the Laramee *et al.* method, called Image Space Advection (ISA) [61], and the Van Wijk method, Image Based Flow Visualization on Surfaces (IBFVS), [111] extend Image Based Flow Visualization [110], a dense texture unsteady two-dimensional 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. Recently, Huang *et al.* extended image-space based visualization to enhance the coherency of the output [45] 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 [64, 49].

Li *et al.* developed Flow Charts for unsteady flow visualization on surfaces [65]. The Flow Chart method decomposes the triangular mesh into patches with a texture atlas, and then the two-dimensional 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 [63, 113, 114]. Finally, this texture is texture-mapped onto the surface during rendering. Flow Charts is a flexible flow visualization scheme, but it has the following drawback: the preprocessing step to decompose the mesh with a particle system is very time consuming.

2.4 Closest Point Method

The closest point method was introduced by Ruuth and Merriman as an embedding surface for solving PDEs [93]. Its usefulness lies 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 two-phase explicit time step as well as evolving a level-set on a surface [72, 71]. März and Macdonald followed up on the work of Macdonald and Ruuth with proofs for the principles of the method [76], and Tian *et al.* followed up on the level-set on a surface with segmentation on a surface [109]. Hong *et al.* applied the closest point method to the level-set equation to simulate fire on an animated surface [44]. Finally, Auer *et al.* used the closest point method to solve the Navier-Stokes equations on dynamic surfaces [2].

2.4.1 Closest Point Grid

The closest point method utilizes the closest point grid, which is similar to a discrete distance field [77, 106], 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.

2.4.2 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

$$v(\mathbf{x}) = u(cp(\mathbf{x})) \Rightarrow \nabla_S u(\mathbf{x}) = \nabla v(\mathbf{x}). \tag{2.2}$$

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, which 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 fashion similar to Eq. 2.2. From these two principles, other differential operators can be constructed, including the Laplace-Beltrami operator [93].

2.5 Sparse Octree

Recently, numerous fast, sparse GPU voxelization for rendering systems have been proposed. GigaVoxels, introduced by Crassin *et al.* [21], renders large volumetric datasets depending on the viewpoint and adaptive data representation. The approach of Laine and Karras [59] is also rendering based, using a slice-based approach to construct a top-down tree. Schwarz and Seidel [97] replaced the two-dimensional rasterization approach previously used with a set of "3D rasterizers", which gave a more flexible scheme by reducing some of the redundant per-triangle processing.

On the other hand, Baert *et al.* [3] proposed a CPU out-of-core sparse voxelization approach. Although not as fast as previous GPU implementations, it is the only method that is not bound by the available memory.

2.6 GPU Computing

Over the last fifteen years, GPU performance has increased signficantly faster than CPU performance in parallel computing. To take advantage of the performance of the GPU, new algorithms have to be developed.

Graphic processing units (GPUs) were historically designed for fast rasterization of three-dimensional graphics. As increasingly advanced hardware was introduced, the fixed functionality of the GPU became a burden as programmers looked to use the GPU for increasingly complex rendering. The Nvidia GeForce 3 was the first commercial consumer graphics card available to support a programmable shading architecture, Microsoft's HLSL 1.0 and DirectX 8. The programmable vertex [66] and fragment shaders, in combination with the ability for fragment shaders to write floating point values to texture buffers, allowed for general-purpose programming in the graphics processing unit, or GPGPU.

The reason to choose GPUs is simple: the parallel floating point processing power. For instance, the Nvidia K80 had approximately 5 to 10 times the double precision FLOPS of an Intel Xeon E5-2697 v2 [85]. Further, this dramatic performance increase has even led to the adoption of GPUs as accelerator cards in exascale computing [25].

2.6.1 Programmable Shaders and GPGPU

Immediately upon the availability of programmable GPUs, one of the first noncomputer graphics-related works was published for fast matrix multiplication on the GPU [62]. From there, using the programming languages available on the GPU at the time, researchers explored the performance of mathematical algorithms on the GPU [58, 32]. At the same time, graphics researchers were also looking at the GPGPU for ray tracing [91] and created stream programming models for the GPU [16]. The stream programming model used in the Brook stream programming language is a powerful abstraction for GPU programming, and many of the ideas can be seen in Thrust, a parallel template library [43]. Thrust provides many of the parallel operators laid out in Brook, including prefix-sum [11] scan [10], reduce, map, sort, filtering, and search. Around this time, there was a significant amount of research done in GPGPU, and we refer the reader to [87] for an overview of the work done.

2.6.2 CUDA

Unfortunately, shading languages were limited by the constraints of the graphics pipeline. In 2007, Nvidia introduced the Common Uniform Device Access (CUDA) language, a dedicated GPGPU programming language that unlocked the performance of the GPU to a wider audience by providing more direct access to the hardware while avoiding the graphics pipeline inherent with the use of shading languages. Although there are other compute languages (OpenCL, DirectX Compute Shaders, OpenGL Compute Shaders, Microsoft DirectCompute), we focus on CUDA and use the terminology from that language.

Logically, the CUDA programming API treats the hardware as a single-instruction multiple-threads (SIMT) architecture. The *host* system runs the program on the *device* that is the physically separate GPU. The hardware is invoked by a kernel, a C/C++ (post CUDA v.7.0) program that is subsequently processed by a large number of threads on the GPU *device*. The number of threads is given at invocation.

The number of threads at execution is a two-level hierarchy, blocks and grids. Blocks are composed of threads that share an execution runtime (and therefore lifetime) and can be synchronized for memory-access purposes. Blocks are grouped into grids. Within the kernel invocation, there are API functions to identify the thread within a block and the block index within the grid, as well as the number of threads in a block and the number of blocks in a grid. Generally, the thread index, in combination with the block index and block dimensions, is used to map to a global memory position to process. Further, the resources available at different hierarchical levels, such as shared memory vs global memory, can be indexed.

There is a memory hierarchy as well. *Global* memory is the largest memory available to the *device*. Typically, this is the video RAM on the GPU. *Global* memory can be accessed as read and write from any thread. *Shared* memory is fast memory with a maximum size of 48KB shared within a single block of threads. *Shared* memory accesss is an order of magnitude faster than *global* memory. Typically, *shared* memory is used as a cache for a thread block to process information it can share between the threads. There are atomic operators available that allow for synchronization at different levels of the hierarchy.

CHAPTER 3

GPU-BASED MESH EXTRACTION

Isosurface extraction from three-dimensional scalar volumes is a fundamental technique in visualization. In some cases, the scalar data may be composed of different materials, and although a material is stored in a regular grid, the materials generally do not conform to the underlying grid. Recent work by Meyer *et al.* [80, 81] uses a particle-based approach to extract curvature-dependent, well-formed multimaterial mesh from biological data. This approach uses an energy-based system to extract a surface mesh with equilateral triangles. Further, it generates meshes with smaller triangles in areas of high curvature, which gives more resolution in areas that need it. Good triangular meshes are an excellent starting point to generate a tetrahedral mesh that is well suited for finite element simulation.

This method generates meshes that are suitable for numerical simulation, but it comes with a very high computational cost. The excessive computational cost to generate a well-shaped multimaterial mesh has hindered the use of the curvature-dependent particle system by the bioengineering community for numerical simulations [108]. Therefore, improving the performance would increase the use of the particle system for various numerical simulation tasks [94].

In recent years, advances in computing power have come from an increase in the number of cores as well as in the frequency of the cores. This increase is true for the graphic processing unit, or GPU, where hundreds of cores are run in a single instruction, multiple thread (SIMT) fashion. To take advantage of this new parallel processing power, efficient parallel algorithms are needed.

3.1 Particle System

The particle system used is based on the dynamic particle system described by Meyer *et al.* [79, 80]. A brief overview of the system is in Fig. 3.1. Initially, a distance field and a sizing field are precomputed to represent the isosurface as an implicit function, F, and to encode the distance between points on F, respectively. Next, particles are seeded on the isosurface based on the results of marching cubes. Then, the particles are processed sequentially: determine neighbors, compute energy and velocity, and update position. A particle moves only if the new position has lower energy than its original position. Once every particle has been processed, the density of the particles is checked to delete or add particles. The above particle process is repeated until the system energy has converged.

3.1.1 Initialization

Before placing the particles, a distance field and a sizing field are precomputed. A distance field of the implicit surface is computed from the scalar data and used with reconstruction filters to generate the implicit function, F [80, 81, 115]. The sizing field, h, is based on the local feature size and *curvature* of the implicit surface and used by the particle system to meet ϵ-sampling distribution requirements [80]. The distance between particles is scaled based on the sizing field in order to control the sampling density, which also reflects the local curvature of the implicit surface (Eq. 3.2). For more information on the construction of the sizing field, see Meyer *et al.* [80]. Once the distance and sizing fields are computed, the system is initialized with a set of particles. The positions of the particles are determined from a marching cubes triangulation to ensure that the entire isosurface is seeded, even the disconnected regions. The initial seeds are then projected onto F (Eq. 3.5).

3.1.2 Per Particle Processing

Processing a particle is a four-step process (Fig. 3.2). First, the neighbors of p_i are determined. Consider all other particles, p_j, in the system where $i \neq j$, p_j is a neighbor of p_i if $\mathbf{d}_{ij} \leq 1.0$, where $\mathbf{d}_{ij}$ is the scaled distance from p_i to p_j. Second, the energy, E_i of p_i, is computed based on its neighbors. Third, the velocity, $\mathbf{v}_i$, at the position of p_i is computed to give a magnitude and direction for p_i to move

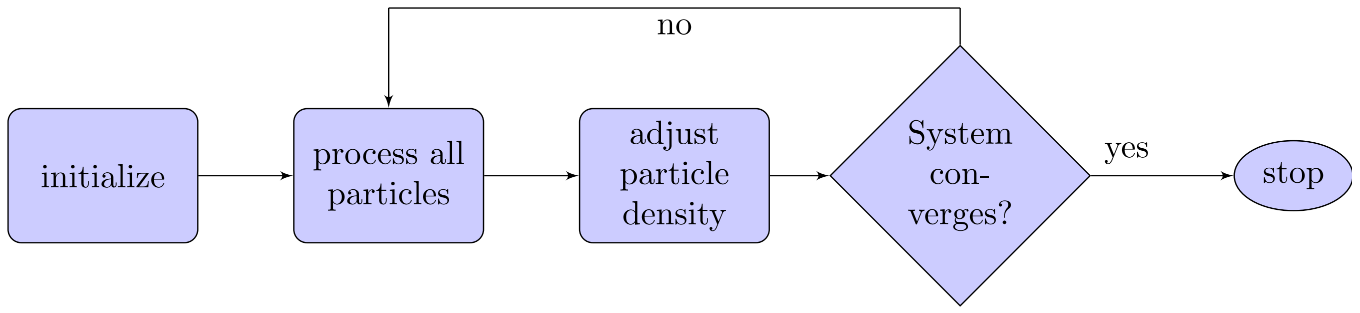


Fig. 3.1. Overview of the particle system.

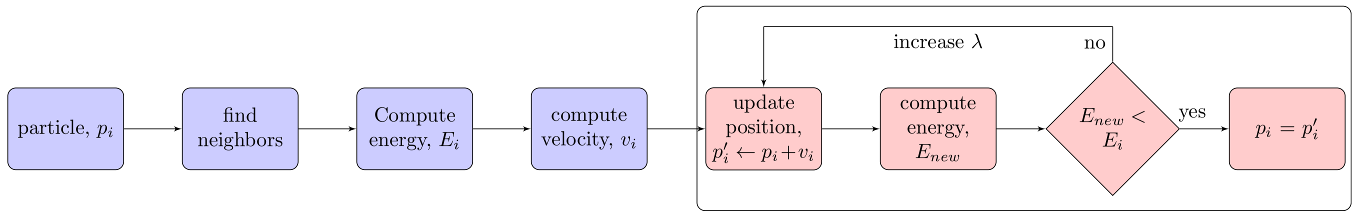


Fig. 3.2. Processing a particle is a four-step process: 1) determine the neighbors, 2) compute the energy, 3) compute the velocity, and 4) update position. The red blocks are the fourth step, i.e., the iterative process to update the position of the particle.

in. Finally, an iterative process (the red blocks in Fig. 3.2) is conducted to update the position of the particle, depending on whether the energy, E_{new}, at the updated particle position $p'_i = p_i + \mathbf{v}_i$, is less than the current energy, E_i. If E_{new} is less than E_i, the particle position is updated to p'_i; otherwise we iterate, with a smaller step size, until the new particle position has a lower energy than the previous position.

3.1.2.1 Energy and Velocity Computation

To compute the energy and the velocity, Meyer *et al.* proposed the cotangent energy function because of its scale invariance and compactness [79]. The energy, E_i, of p_i is the sum of the energies E_{ij} between p_i and p_j such that

$$E_{ij} = \begin{cases} cot(|\mathbf{d}_{ij}|\frac{\pi}{2}) + |\mathbf{d}_{ij}|\frac{\pi}{2} - \frac{\pi}{2} & |\mathbf{d}_{ij}| \leq 1.0 \\ 0 & |\mathbf{d}_{ij}| > 1.0 \end{cases} \tag{3.1}$$

and

$$\mathbf{d}_{ij} = \frac{|p_i - p_j|}{2 \times cos(\frac{\pi}{6}) \times min(h_i, h_j)} \tag{3.2}$$

where $\mathbf{d}_{ij}$ is the scaled distance between p_i and p_j ($i \neq j$) and $|p_i - p_j|$ is the Euclidean distance between particles p_i and p_j. We will refer to $\mathbf{d}_{ij}$ as the distance between p_i and p_j, in the rest of this chapter.

To compute distance, $\mathbf{d}_{ij}$, between p_i and p_j, the sizing values, h_i and h_j, at p_i and p_j are used (Eq. 3.2). The distance between p_i and p_j is scaled by the $min(h_i, h_j)$. Because the distance and energy are scaled by the surface curvature as in Eq. 3.2, when the distance is less than 1.0 (i.e., within the neighborhood of the desired radius), the energy E_{ij} is computed between the two particles using Eq. 3.1. Otherwise, there is no energy between them and $E_{ij} = 0$.

The energy of a particle is used to determine whether a new position, p'_i, is at a lower energy state than the original position. However, to move p_i, the velocity of p_i is computed. The velocity, $\mathbf{v}_i$, is the derivative of the energy function. The velocity for p_i is computed as the sum of all the velocities, $\mathbf{v}_{ij}$, between p_i and p_j and ($i \neq j$) where

$$\mathbf{v}_{ij} = -(\tilde{H}_i)^{-1}\left(\frac{\partial E_{ij}}{\partial|\mathbf{d}_{ij}|}\frac{\mathbf{d}_{ij}}{|\mathbf{d}_{ij}|}\right) \tag{3.3}$$

and

$$\frac{\partial E_{ij}}{\partial|\mathbf{d}_{ij}|} = \begin{cases} \frac{\pi}{2}\left[1 - sin^{-2}(|\mathbf{d}_{ij}|\frac{\pi}{2})\right] & |\mathbf{d}_{ij}| \leq 1.0 \\ 0 & |\mathbf{d}_{ij}| > 1.0 \end{cases} \tag{3.4}$$

where $\tilde{H}_i$ is the Hessian of p_i's potential with the diagonal of $\tilde{H}_i$ adjusted by λ according to the Levenberg-Marquardt (L-M) algorithm. The L-M algorithm is discussed further in Section 3.1.2.2. The velocity is used to move p_i in the tangent plane of the F at p_i. Once p_i is moved in the tangent plane, it is projected back onto the surface,

$$p_i \leftarrow p_i + F_i \frac{\nabla F_i}{\nabla F_i \cdot \nabla F_i} \tag{3.5}$$

where F_i is the implicit function and ∇F_i is the gradient of the implicit function at p_i.

3.1.2.2 Update Position

Updating the position of the particle is an iterative process to find the appropriate step size for $\mathbf{v}_i$. The L-M algorithm is used because with the current step size of $\mathbf{v}_i$, the particle may not be moved to a place with lower energy. Each particle has a λ value, which it maintains throughout the entire run of the particle system. Increasing λ decreases the step size of $\mathbf{v}_i$. As λ is increased (or decreased), the step size of $\mathbf{v}_i$ is converging to a good step size, i.e., the step will produce a proper velocity that leads to a lower energy state. In practice, λ is incremented by 10. For more details on the L-M algorithm, see Meyer *et al.* [79].

Algorithm 3.1 is used to update the position of p_i. A possible new position, $p_i' = p_i + \mathbf{v}_i$, is computed. The energy of p_i', E_{new}, is computed using Eq. 3.1. If $E_{new} < E_i$ then p_i is updated to its new position p_i'. Otherwise, the particle system iteratively increases λ and computes a new particle position $p_i' = p_i + \mathbf{v}_i$ and energy, E_{new}, until $E_{new} < E_i$ or $\lambda \geq \lambda_{max}$. If $\lambda \geq \lambda_{max}$, then the particle's position is

Algorithm 3.1 Update Particle Position

```
$iterate \leftarrow true$
**while** iterate **do**
    increase $\lambda$ by 10
    $p'_i \leftarrow p_i + v_i$
    Project $p'_i$ onto surface.
    **for all** particles $p_j$ in neighborhood NH **do**
        **if** $p'_i \neq p_j$ **AND** $distance(p_i, p_j) \leq 1.0$ **then**
            $E_{ij} \leftarrow$ **calcEnergy() as in Eq. 3.1**
        **end if**
    **end for**
    $E_{new}$ **= sum** $E_{ij}$ **over** $NH$
    **if** $E_{new} < E_i$ **then**
        **Save** $\lambda$
        $p_i \leftarrow p'_i$
        $iterate \leftarrow false$
    **else if** $\lambda \geq \lambda_{max}$ **then**
        $iterate \leftarrow= false$
        **reset** $\lambda$ **to its original value.**
    **end if**
**end while**
```

not updated, and λ is reset to its value at the beginning of the iteration process. Otherwise, the position of p_i is updated to p'_i.

3.1.3 Density Control

Controlling the density of the particles is an important aspect in the placement of the particles. Recall that the particle system is initially seeded with particles on the surface from marching cubes. However, the number of particles needed to create the proper density is not known a priori. Therefore, we may seed too many or too few particles. If that is the case, no matter how the particles are moved, an optimal configuration may not be achieved.

Therefore, at the end of every iteration, the energy, E_i, of every particle p_i is checked against an ideal energy, E_{ideal}. Recall that E_i is calculated from the distance, $\mathbf{d}_{ij}$, of p_i to its neighbors, p_j and $\mathbf{d}_{ij}$ is adjusted by the sizing field, h_i (Eq. 3.2). If the energy is too high, there are too many particles close to p_i. If the energy is too low, then there are not enough particles close to p_i. The ideal energy of a particle, $E_{ideal} = 3.462$, is based on the energy computed from a natural hexagonal configuration [79]. In other words, the desired configuration is to have six neighboring particles. Achieving E_{ideal} is controlled through the addition and deletion of particles. The addition or deletion of particles is biased with a random value from $[0,1]$ to prevent mass addition or deletion [116].

3.1.4 Binning and Neighborhoods

The complexity of the aforementioned particle system as explained is $O(N^2)$. A particle's energy and force are determined by the distance to every other particle in the system. Heckbert introduced binning as an acceleration structure [42]. Instead of computing energy between a particle and every other particle in the system, he subdivided the space according to a parameter, σ. Thus, it was only necessary to compare a particle with its immediate neighbors. By setting the bin length to at least σ, it is guaranteed that all possible neighbors are located within the current bin plus all the surrounding bins, i.e., the neighborhood. The neighboring bins must be included since a particle may lie near the edge of the bin, and therefore its neighbors would be in the surrounding bins. Because the sizing field contains the distance between particles needed for a quality reconstruction, it is used to determine the bin size as $\sigma = max(h)$, the global maximum of the sizing field. This acceleration structure is used to speed up the particle system described by Meyer *et al.* and is implemented in BioMesh3D.

3.2 Parallelization

This GPU parallel implementation relies on neighborhoods to allow concurrent processing of bins at the CUDA block level. Whereas particles are advected concurrently, the computation of the energy E_i of particle p_i and the velocity $\mathbf{v}_i$ of p_i are dependent on the distance d_{ij} from p_i to its neighbors p_j. This computation is parallelized as well, where a CUDA thread is assigned to each pairwise computation, p_i to each of its neighbors p_j. This parallelization strategy maximizes the amount of work done, but also gives the flexibility required due to the uncertain nature of determining the velocity step size.

3.2.1 Bin Processing

Instead of trying to process all of the particles concurrently, groups of particles can be processed simultaneously if their neighborhoods do not overlap. The binning structure provides the necessary knowledge for such a grouping since every particle contained in a bin is a potential neighbor to every other particle within the same bin. To guarantee a correct energy and velocity computation, the particles in the bins neighboring the current bin are also considered as neighbors of every particle in the current bin. That said, the particles in the neighboring bins cannot be processed simultaneously while the particles in the central bin are being processed. Therefore, no overlapping neighborhoods are allowed for any groups of particles that are being processed concurrently. Before attempting to run groups of particles concurrently, though, how the particles are processed needs to be changed. Previously, all particles in the system were processed serially as described in Fig. 3.1. Instead, since the particles are binned, the particle system can process the groups of particles. Thus, for each bin, B, and its neighborhood, NH, in the particle system, all the particles $p_i \in B$ are processed serially as shown in Fig. 3.3. Although this change does not affect serial processing of the particles within a bin, it allows particles to be processed concurrently by executing bins with nonoverlapping neighborhoods.

If the particles are grouped (and processed) by their bins, the bins can be processed in parallel but only if the neighborhoods do not overlap. Recall that the bin size is $max(h)$. The step size is limited to a maximum of the sizing field, h, which means the particle can travel into an adjacent bin. Therefore, given a bin $B(a,b)$ and its neighborhood, $NH = \bigcup_{i=a-1,j=b-1}^{i=a+1,j=b+1} B(i,j)$, if $B(a,b)$ is currently processed, the other bins that can run concurrently are $B(a+3k, b+3m)$. An example of processing multiple bins concurrently is given in 2D in Fig. 3.4. The bins in Fig. 3.4a that are about to be processed are labeled W through Z. Bin W is at position (0,0); therefore, the next bins that are processed concurrently are at positions (3,0), (0,3) and (3,3) for X, Y, and Z, respectively. Once all the particles in bins W through Z have been processed, the next bins are processed as in 3.4c and 3.4d. This procedure is repeated until all the bins in the 3×3 space, i.e., the compute block, are processed.

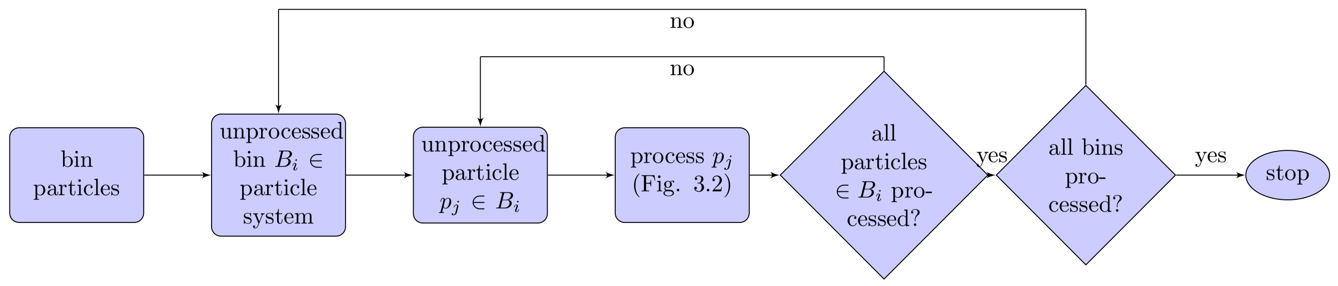

bin particles
unprocessed bin $B_i \in$ particle system
unprocessed particle $p_j \in B_i$
process p_j (Fig. 3.2)
all particles $\in B_i$ processed?
all bins processed?
stop
yes
yes
no
no

Fig. 3.3. Processing particles by their bins.

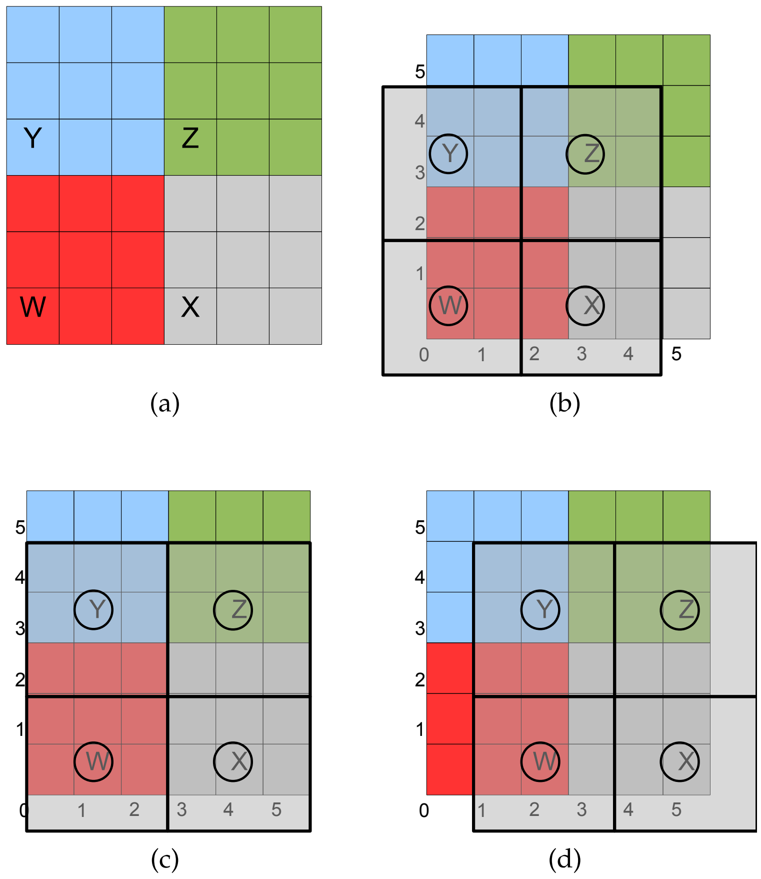


Fig. 3.4. Running multiple neighborhoods concurrently in 2D. (a) Bins to be processed are labeled. (b) Neighborhoods are highlighted. (c) Move to next bins. (d) Move to next bins.

3.3 CUDA Implementation

In the previous sections, we described how particles are moved and how bins can be run concurrently. Now, we explain how the particle system is run on the GPU. The motivation for using the GPU is simple. Recently, processing power on the GPU has outstripped the CPU [86]. Further, parallel computing architectures such as CUDA have made that processing power more accessible than what was previously available with GPU shaders alone. Although the GPU has more processing power than the CPU, it also has limitations. In particular, the GPU is a massively parallel system with many hardware threads. Unfortunately, these hardware threads do not handle divergence well, where control statements may cause threads to follow different execution paths, which serializes the computations [86]. With the use of the Levenberg-Marquardt algorithm (L-M), it is not possible to run a particle per thread because there is no way to know a priori how many iterations the L-M algorithm will take to find an appropriate velocity step size. If every particle requires a different number of iterations to determine the step size, all the threads would have to run serially, which hinders performance. Beyond the thread divergence limitation, memory management is important as well. In particular, coalescing memory fetches is very important. Coalesced memory fetches require memory to be aligned when fetched from global memory.

With divergence and memory management in mind, running the particle system on the GPU is as follows. First, bins are run concurrently (Section 3.2.1) by processing a bin in a CUDA thread block because processing a bin per thread is not possible due to thread divergence. Second, note that a thread block is composed of tens to hundreds of CUDA threads, so for every particle run in a thread block, multiple threads are available for processing. Thus, the pairwise energy and velocity computations can be processed in parallel. Finally, memory management is discussed. To coalesce memory access, neighborhoods are copied into contiguous memory. Further, preprocessed data, i.e., the sizing and distance fields, use texture memory for automated memory management.

3.3.1 Bin Processing

Bins are processed concurrently by executing a CUDA block per bin. Assign each bin B_i and its neighborhood (see Fig. 3.4) to a CUDA block CB_i. Processing all the bins in the particle system means iteratively processing bins in a compute block. Thus, once a group of bins is processed, the adjacent bins are processed next. We continue until all the bins have been processed, as illustrated in Fig. 3.4. This is the block-level parallelization.

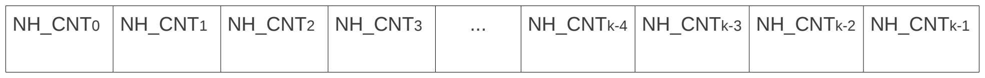
3.3.2 Energy and Velocity Computation

Since a thread block is run per bin and particles are run serially within a bin, when $p_i \in B$ is processed, multiple CUDA threads are used to calculate the energy and velocity. A CUDA thread, t_j, is assigned to do the pairwise energy computation from p_i to one other $p_j \in NH$. Once the pairwise energy calculations are finished, a parallel sum reduction is conducted to compute E_i from the array of energy values, E_{ij}. The velocity is computed in a similar manner to the energy computation. By running a CUDA block per bin, the computation is parallelized at both block (bins) and thread (energy and velocity computation) levels.

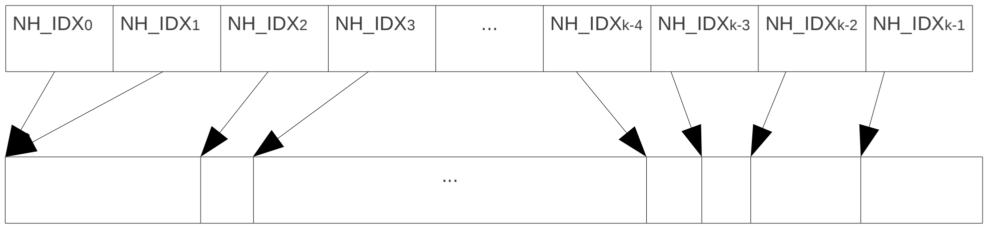
3.3.3 Memory Management

The method to build the bins efficiently in CUDA is similar to the one used to build spatial subdivision for uniform grids in Green [40]. To coalesce memory access, at the beginning of every iteration the indexes of the particles are binned in global memory. Additionally, a particle count is generated for every bin, B_CNT. Before each neighborhood is processed, the particles are copied into a contiguous span of global memory. As p_i is processed serially in bin B, and the energy (or velocity) is computed according to Eq. 3.1 (or Eq. 3.3), a thread, t_j, is assigned for the pairwise computation. Copying the particles to coalesce memory access constitutes less than 4% of the total run-time required.

To create multiple neighborhoods, NH_k, in global memory, NH, compactly and concurrently, a three-step approach is used as outlined in Algorithm 3.2. First, the number of particles in each NH_k are counted (Fig. 3.5a). For each NH_k, and for each bin $B_i \in NH_k$, $NH\_CNT_k \mathrel{+}= B\_CNT_i$. Second, the particle system computes the



(a) Number of particles per bin



(b) Index into neighborhood array

Fig. 3.5. Memory layout in CUDA.

Algorithm 3.2 buildNeighborhoods()

```
for all neighborhoods $NH$ do
    for all bins $B \in NH$ do
        $NH\_CNT$ += num of particles $\in B$
    end for
    $NH\_IDX_k$ = atomicInc(ptr, $NH\_CNT$)
    Copy particles in $NH$ to $NH\_IDX$
end for
```

memory location, NH_IDX_k of NH_k (Fig. 3.5b). Recursively, it is defined as

$$NH\_IDX_k = NH\_IDX_{k-1} + NH\_CNT_k \tag{3.6a}$$

with

$$NH\_IDX_0 = 0. \tag{3.6b}$$

To determine the neighborhoods concurrently in CUDA, Eq. 3.6, the CUDA *atomicInc*() function and a global integer, *ptr*, are used to create the array of indexes. The *atomicInc*() function takes two values, a memory reference *ptr* and an integer *val*, and returns the previous value, *prev*, at P atomically. Thus, although every neighborhood in the particle system is calling *atomicInc*(), it is serialized because the *ptr* can be incremented only by NH_CNT_k atomically. Therefore, $NH\_IDX_k = ptr + NH\_CNT_k$ where $ptr = NH\_IDX_{k-1}$. Third, with an index, NH_IDX_k into the span of global memory reserved for NH, it is easy to copy particles into their respective neighborhoods (Fig. 3.5b). This procedure produces a per neighborhood count of particles for each neighborhood, a per neighborhood index into the list of particles, and a copy of all the particles binned into their neighborhoods. As mentioned before, this procedure is done to copy a neighborhood into contiguous memory to coalesce memory access.

The sizing field is precomputed in a separate process, and therefore the data is read into a three-dimensional texture to take advantage of texture caching. However, the built-in interpolation function was not accurate enough. The hardware trilinear interpolation is only a "9-bit fixed point format with 8 bits of fractional value" [86]. Instead, a full float type trilinear interpolation function was used. Every thread block has a shared memory variable for the sizing field value at its location for better localized access. Likewise, the distance field is precomputed and read into a texture for the same reasons the sizing field was put into a texture. However, instead of trilinear interpolation, cubic B-spline kernels were used to reconstruct the surface and its gradient and Hessian [80]. Although slower than trilinear interpolation, the cubic B-spline kernel balances efficiency with good derivative approximations [82].

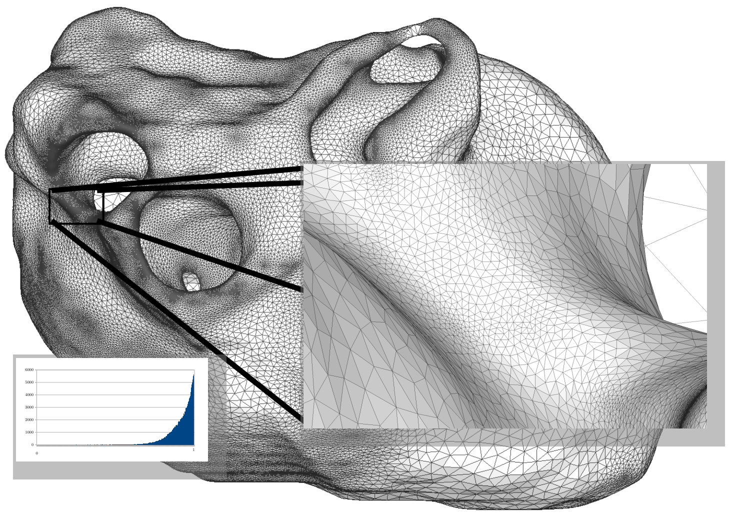
Finally, because of the addition and deletion of particles, the particles are double buffered between iterations. The addition or deletion of a particle is carried out after all the particles have been processed. If the energy of the particle is not within a certain threshold of E_{ideal}, then it is either added or deleted. In practice, if $E_i < .75 \times E_{ideal}$, then a particle is added, and if $E_i > 1.35 \times E_{ideal}$, then the particle is deleted. The energy calculation for adding or deleting particles is done in the same manner as moving the particles, with the block-level and thread-level parallelization. Although adding or deleting can be performed without the double buffer, this helps cluster the particles by region and allows for faster binning in the next iteration.

3.4 Results

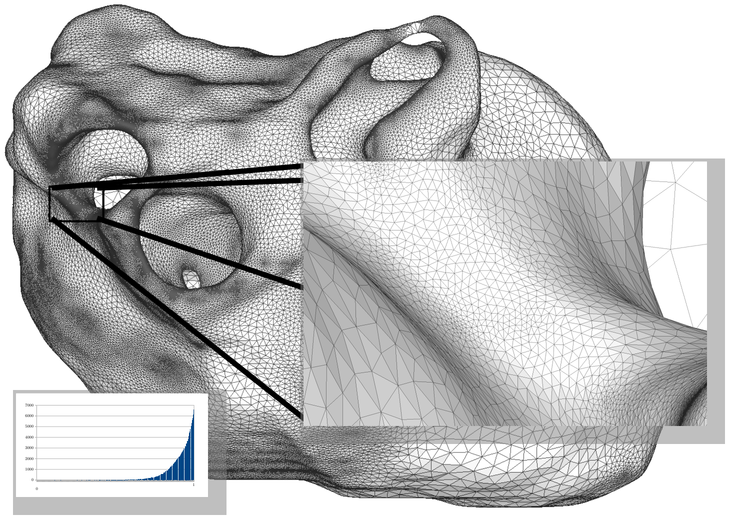
A CPU version of the particle system, BioMesh3D [98], is used to generate the CPU mesh. A level set method [115] is used to generate the distance field and the sizing field h in the precomputation step. A B-spline reconstruction kernel is used to interpolate values and compute the gradient and the Hessian of F. For the sizing field, h, linear interpolation is used to lookup the values at p_i.

Once the particles have been saved from BioMesh3D or the CUDA implementation, TIGHT COCONE [24] is used to create a water-tight mesh. The three-dimensional scalar fields are 268x129x177 volume data of a human heart, human lungs, and human ribcage. The results of the heart, lungs, and ribcage (CPU and GPU) are in Figs. 3.6a through 3.8b.

Marching cubes is used to seed the particles and is generated on the CPU. Once the initial particles are seeded and projected onto the surface, they are copied to the GPU, and the system processes the particles as described in the previous sections. Once 50 iterations are completed or the energy has stagnated where $\frac{E_{prev} - E}{E} < E_{min}$, the process is terminated. We have found in practice that $E_{min} = 0.0015$ produces good meshes. All tests were run on an nVidia Tesla c2070 with 6GB of RAM and an Intel Xeon X5650 2.67Ghz with 196GB of RAM.

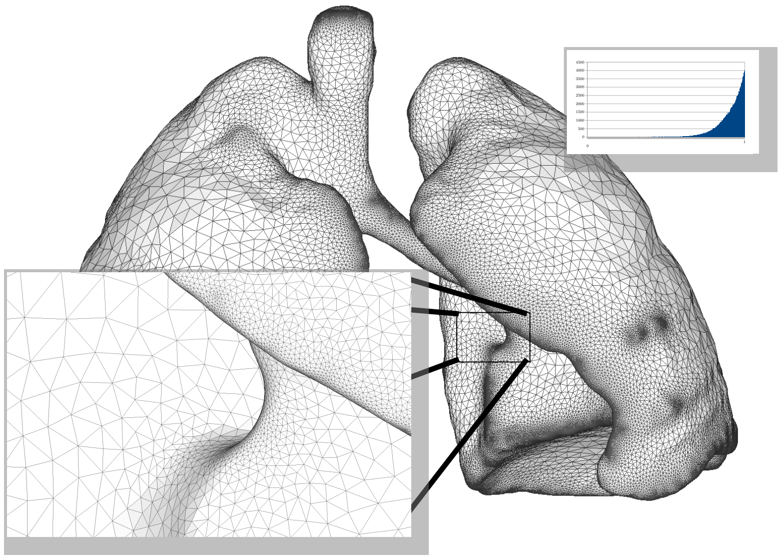


(a) GPU heart with zoomed-in image and histogram of radius ratio.



(b) CPU heart with zoomed-in image and histogram of radius ratio.

Fig. 3.6. Images of the heart dataset on the CPU and GPU, respectively. Further, embedded is a zoomed-in area of the image and the histogram for the dataset. The visual quality of the CPU implementation compared to the GPU implementation is very similar. Further, the histograms show that both the CPU and GPU systems are dominated by well-shaped triangles.


(a) GPU lungs with zoomed-in image and histogram of radius ratio.

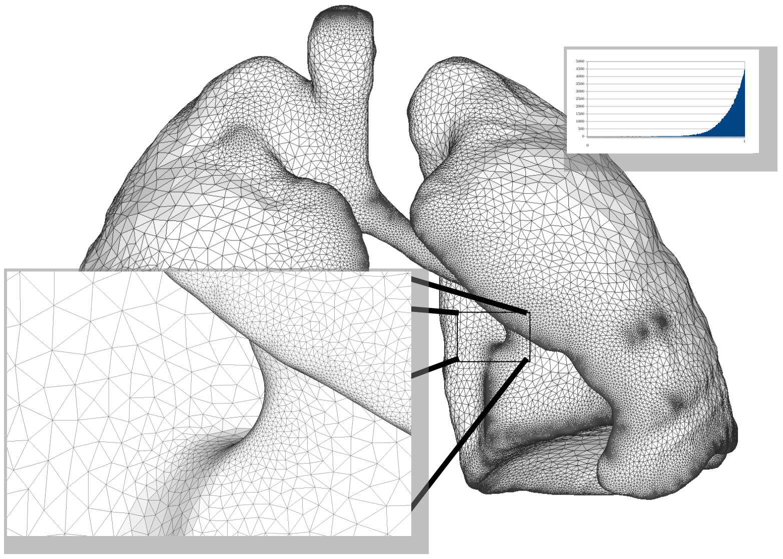
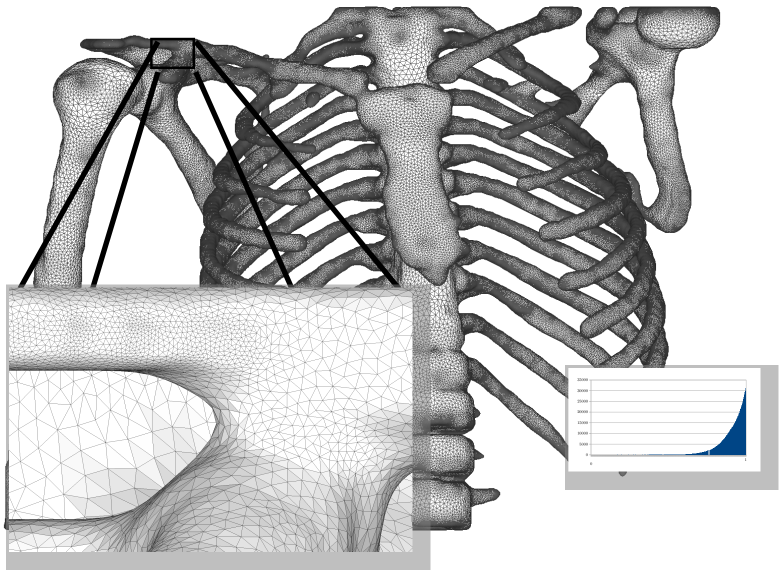
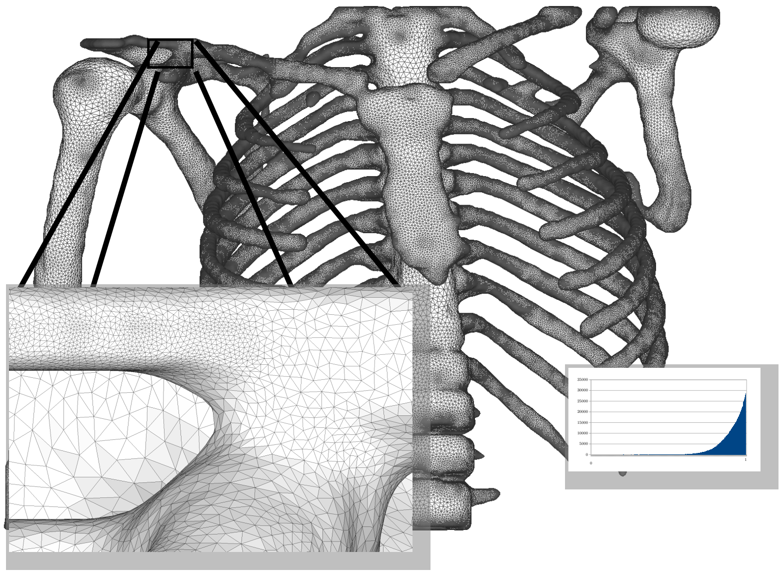

(b) CPU lungs with zoomed-in image and histogram of radius ratio.

Fig. 3.7. Images of the lung dataset on the CPU and GPU, respectively. Further, embedded is a zoomed-in area of the image and the histogram for the dataset. The visual quality of the CPU implementation compared to the GPU implementation is very similar. Further, the histograms show that both the CPU and GPU systems are dominated by well-shaped triangles.



(a) GPU torso with zoomed-in image and histogram of radius ratio.



(b) CPU torso with zoomed-in image and histogram of radius ratio.

Fig. 3.8. Images of the ribcage dataset on the CPU and GPU, respectively. Further, embedded is a zoomed-in area of the image and the histogram for the dataset. The visual quality of the CPU implementation compared to the GPU implementation is very similar. Further, he histograms show that both the CPU and GPU systems are dominated by well-shaped triangles.

3.4.1 Quality

To evaluate the quality of the obtained mesh, the ratio of the inscribed and circumscribed radii is computed for every triangle on the mesh, and the mean radius ratio of the mesh is calculated. The higher the ratio between inscribed and circumscribed radii, the closer a triangle is to being equilateral. The radius ratio is a common quality metric that allows a direct comparison between two meshes.

Table 3.1 includes the qualitative results. The mean ratio of a mesh generated through the GPU system is within 1% of the mean radius ratio of the CPU implementation. Thus, the GPU meshes have a very similar quality to the CPU meshes. The histograms in Fig. 3.6a through 3.8b generated for the heart, lungs, and ribcage, respectively, show that the distributions of the ratios are dominated by good triangles and that both the CPU and GPU meshes have similar profiles. The close-up images in Figs. 3.6a through 3.8b show that the quality of the mesh using our GPU particle system is similar to or comparable to the one using the CPU version.

3.4.2 Speed-up

The quality of the meshes is nearly the same, but there is a substantial performance gain with the GPU version (Table 3.2). The GPU version is 7.8× to 35.2× faster than the single-threaded CPU implementation. The reductions in the run-time are from 835.26 to 107.64 seconds for the lungs, 3150.38 to 245.77 seconds

TABLE 3.1. Multiple datasets, including heart, lungs, and ribcage on the CPU and GPU, are compared for quality. Qualitative comparison is done by calculating the mean radius ratio of the resulting meshes.

	CPU		GPU	
Dataset	Rad. Ratio	Min. Ratio	Rad. Ratio	Min. Ratio
Heart	0.92114	0.249245	0.92079	0.117757
Lungs	0.912578	0.217819	0.913214	0.324375
Ribcage	0.914975	0.186664	0.914975	0.186664

TABLE 3.2. The amount of time to place particles on the surface is compared in this table. Multiple datasets, including heart, lungs, and ribcage on the CPU and GPU, are listed along with the time, in seconds, to place the particles and the final number of particles for the CPU and the GPU, respectively. The last column is the speed-up gained from the GPU system.

	CPU		GPU		
Dataset	Time	# Particles	Time	# Particles	Speed-up
Lungs	835.26	74153	107.64	74129	7.8x
Heart	3150.38	80125	245.77	80594	12.8x
Ribcage	9460.29	468877	269.12	468623	35.2x

for the heart, and 9460.29 to 269.1 seconds for the ribcage (Table 3.2). Those are 7.8, 12.8, and 35.2× speed-up of the GPU over the CPU, respectively.

3.4.3 Scaling

In the previous section, there was a correlation between the number of particles and the speed-up. As the number of particles increases, so does the speed-up, but this is across different implicit functions. To measure the speed-up, we conducted a real-world test and a synthetic test using the ribcage dataset. The real-world test controls the number of particles by varying ϵ and δ parameters when generating the sizing field around the isosurface. The ϵ and δ parameters control the density of the particles, where the smaller the values of ϵ and δ, the denser the particles [80].

However, for the ribcage dataset, the lowest number of particles generated by manipulating the ϵ and δ values in the precomputed phase was 320,000. Generating a sizing field using $\epsilon > 8.0$ and $\delta > 2.0$ resulted in an incomplete mesh. For instance, with $\epsilon = 10.0$ and $\delta = 5.0$, the ribs of the ribcage were removed. Therefore, a synthetic test was created. The synthetic test removes the *add new particles* stage and seeds a user-defined number of particles, which creates an upper bound on the number of particles in the system. This seeding is done through marching cubes and generates an initial seeding that is closer to the original implicit function than using large ϵ and δ values.

For the synthetic test, the seed numbers were 60,000 to 300,000, increasing by 30,000. Note in Table 3.3 that although adding particles is disallowed, removing particles is still active. Therefore, the final particle count is less than the initial number seeded. Fig. 3.9 shows a plot of the amount of time to generate a mesh versus the number of particles. As the number of particles increase, the speed-up increases as well, from 6.14× speed-up of the GPU over the CPU with 57,000 particles to 22.0× speed-up with 230,000 particles. Therefore, for the synthetic test, as the number of particles increases, the speed-up increases in a linear manner.

The synthetic test is useful to verify linear speed-up when the number of desired particles is not achievable by changing the sizing field, but the real-world test is a better reflection of attainable speed-ups. Table 3.4 contains the data from generating different sizing fields dependent on the ϵ and δ values. Further, the iteration number is the number of times the level set method is run to generate the sizing field. Thus, the more iterations of the level set method, the denser the particles.

The real-world test mirrors the results of the synthetic test, i.e., the speed-up is related to the number of particles. Fig. 3.10 is a graph of Table 3.4 comparing the GPU (in blue) timing results in seconds versus the CPU (in red) timing results. Fig. 3.11 is an image of the real-world ribcage dataset and the embedded images

TABLE 3.3. Synthetic test data for scaling the ribcage dataset without adding any particles to give an upper bound on the number of particles. The details are the initial number of particles (60,000 to 300,000), the time and final number of particles for the CPU system, and the time and final number of particles for the GPU system and the speed-up.

	CPU		GPU		
Init. Parts.	Time	# Particles	Time	# Particles	Speed-up
60000	213.22	57456	34.75	56844	6.14x
90000	444.62	81432	48.82	80208	9.1x
120000	756.8	103913	66.35	103716	11.4x
150000	1360.87	131145	98.26	133792	13.8x
180000	1571.96	145958	100.76	146754	15.6x
210000	2354.04	170805	141.4	175775	16.6x
240000	2860.53	185035	160.28	194354	17.8x
270000	3455.14	200925	172.31	208866	20.1x
300000	4042.60	225054	183.98	237921	22.0x

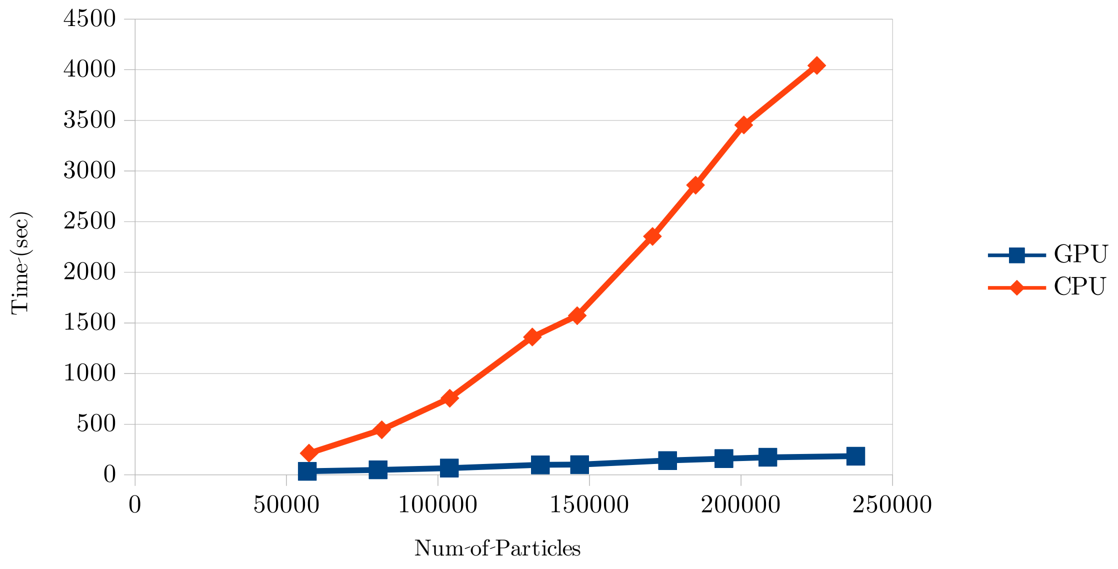


Fig. 3.9. Synthetic test for the ribcage dataset. Graph of Table 3.3 where the red plot is the CPU and the blue plot is the GPU.

TABLE 3.4. Real-world test data for scaling the ribcage dataset by varying the ϵ and δ when generating the distance field. The fields are the ϵ, δ and iteration count used to generate the sizing field, the time and final number of particles for the CPU implementation, the time and number of particles for the GPU implementation, and the speed-up of the GPU system over the CPU system.

			CPU		GPU		
ϵ	δ	Iterations	Time	# Particles	Time	# Particles	Speed-up
8.0	2.0	4	3798.74	317809	160.17	323762	23.7x
2.0	1.0	4	5526.21	377681	200.6	384531	27.6x
0.5	0.5	4	5952.6	398838	212.19	405097	28.1x
0.25	0.25	4	8805.9	428885	265.8	431491	33.1x
0.125	0.125	4	9460.29	464265	269.12	468623	35.2x
0.01	0.01	4	11356.3	493697	285.51	494477	39.8x
0.01	0.01	7	19750.2	530565	445.49	530717	44.3x

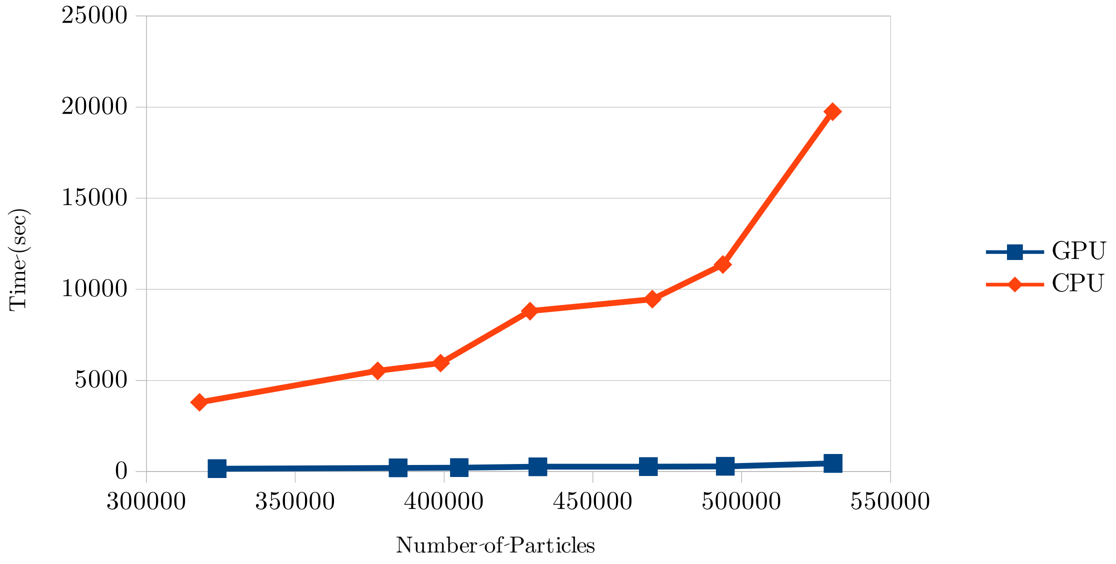
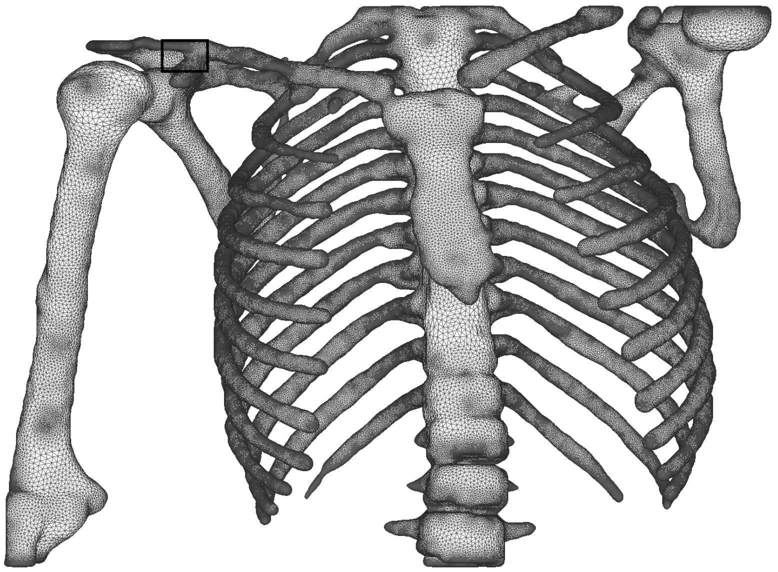
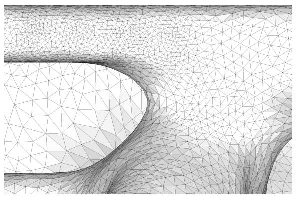


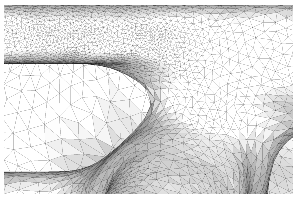
Fig. 3.10. Real-world test for the ribcage dataset. Graph of Table 3.4 timing results as the number of particles are increased. The GPU results are in blue and the CPU results are in red.



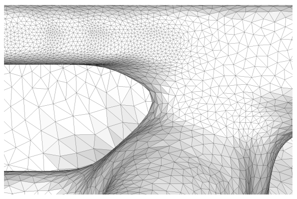
(a) $\sigma = 0.125$, $\delta = 0.125$, with the area marked for Fig. 3.11c - 3.11d



(b) $\sigma = 2.0$, $\delta = 1.0$, 384,531 particles



(c) $\sigma = 0.5$, $\delta = 0.5$, 405,097 particles



(d) $\sigma = 0.125$, $\delta = 0.125$, 468,623 particles

Fig. 3.11. Three meshes of the same dataset, with varying number of particles. As the σ and δ parameters are decreased, the number of particles increases.

are selected ϵ and δ values. As the ϵ and δ parameters are decreased and the iteration number is increased, the number of particles increases while the speed-up increases as well (Fig. 3.10). Further, as the number of particles increases, the speed-up increases in a linear manner as well. For the CPU results in Fig. 3.11, it appears increasing the number of iterations of the level-set method from 4 to 7 increases the amount of time in a nonlinear fashion. This indicates that, for the CPU implementation, the binning structure performance is limited by the number of particles and will regress to $O(n^2)$.

3.5 Summation

This chapter described a new method to parallelize particle meshing by processing bins concurrently. Further, on the GPU, by mapping bins to thread blocks, the energy and velocity computations are parallelized as well. We have presented a variety of datasets that show a reliable speed-up can be achieved regardless of the number of particles. We compared the accuracy of the GPU particle system against a CPU particle system and demonstrated that the resulting meshes are similar as measured by the mean ratio of the triangles. Finally, we have shown that as the number of particles increases, so does the speed-up of the GPU over the CPU.

This chapter explained the adaption of a single distance field to the GPU. The next section discusses adapting multiple material mesh extraction on the GPU.

3.6 Multimaterial Mesh Extraction on the GPU

The multimaterial surfacing in this work is based on the multimaterial surfacing of Meyer *et al.* [81]. Their multimaterial implementation is based on functional representation from [89] where the interfaces are modeled by a function. The interface model is a set of N *indicator functions* $F = \{F_i | f_i : V \longmapsto R\}$, which represents N materials. A material label is assigned to a point $x \in V \iff f_i(x) < f_j(x) \forall i \neq j$. In this multimaterial model, a *junction* is the set of all points x, where $f_i(x) - f_j(x) = 0$. Generally, for three dimensions, there are three types of junctions: four material junctions that are points, three junctions that are lines, and two junctions that are surfaces.

The *cell indicator function*, J, approximates the material junction between different materials and starts with an *inside/outside function*,

$$\tilde{f}_i = f_i - max^n_{j=1, j\neq i} f_j \tag{3.7}$$

where the function *max* is the maximum value of $f_j, \forall j \neq i$. The inside/outside function has a value greater than zero when in the presence of material i and a value less than zero for other materials and a value of zero when on the zero set. With the zero set between adjacent materials defined as $\tilde{f}_i = \tilde{f}_j = 0$, the *cell indicator function*, J_i is defined as

$$J_{ij} = \tilde{f}_i^2 + \tilde{f}_j^2 \tag{3.8}$$

for the points on the zero set between the two materials f_i and f_j. The gradient of the two material junction, Eq. 3.8, is

$$\nabla J_{ij} = 2\tilde{f}_i \nabla \tilde{f}_i + 2\tilde{f}_j \nabla \tilde{f}_j. \tag{3.9}$$

To distribute particles over the junctions, the particles are advected in the local tangent space of the manifold and reprojected back onto the surface. Both steps require first derivative information. However, the function max in Eq. 3.7 is only C^0, which makes max undefined on the junction. An approximation is used for *max* [81],

$$max(V) = \frac{1}{2^{m-1}} \sum_{i=1}^{m} v_i \prod_{j=1, j\neq 1}^{m} g(v_i - v_j) \tag{3.10}$$

where m is the number of functions and $g()$ is a differentiable copysign function [81]. The gradient of the function *max* is

$$\nabla max(V) = \frac{1}{2^{m-1}} \sum_{i=1}^{m} \left[\nabla v_i \prod_{j=1, j\neq i}^{m} g(v_i - v_j) + v_i \left(\sum_{j=1, jnei}^{m} \nabla g(v_i - v_j) \sum_{j=1, j\neq i}^{m} g(v_i - v_j) \right) \right] \tag{3.11}$$

where $\nabla g()$ is the derivative of the copysign function [81].

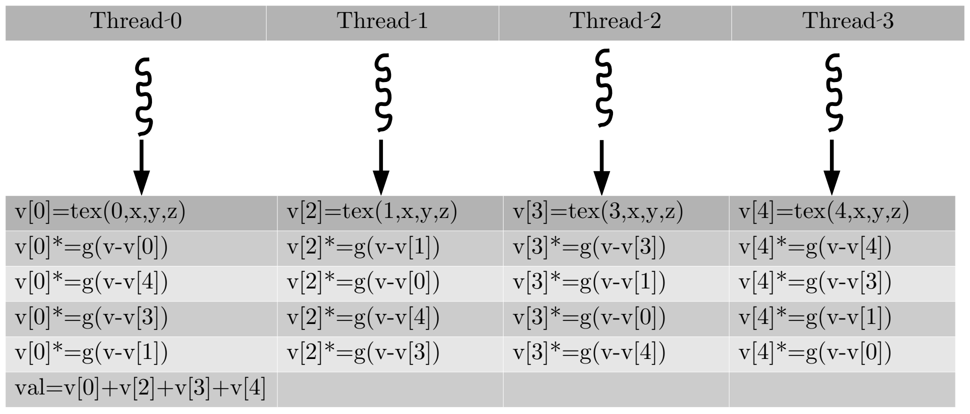
Finally, the gradients of the inside/outside function will be close to equal and opposite near the zero set. The projection vector is defined as the average of these inside/outside functions such that

$$\mathbf{n}_t = \frac{\nabla \tilde{f}_i - \nabla \tilde{f}_j}{| \nabla \tilde{f}_i - \nabla \tilde{f}_j |}. \tag{3.12}$$

One key observation is centered on optimizing the distance field and sizing field interpolations for performance on the GPU. Optimizing the interpolation is particularly important because the distance and the sizing field values at particle p_i are recalculated anytime p_i *potentially* moves. In other words, finding the correct step size requires temporarily placing the current particle p_i at a new position, $\bar{p}_i$ and calculating its temporary energy $\bar{E}$. However, calculating $\bar{E}$ requires the sizing field value, $\bar{h}$, at $\bar{p}_i$. Further, the projection of $\bar{p}_i$ after its step in the tangent plane may require a gradient search before $\hat{p}_i$ is back onto the surface.

First, the precomputed distance and sizing fields are loaded into three-dimensional textures on the GPU. When a particle, p_i is moved in the system, *max* (Eq. 3.10) and ∇*max* (Eq. 3.11) are called numerous times to advection the particle and project it back onto the surface. Both functions are parallelized dependent on the number of materials. Fig. 3.12 is an example of the *max* function, given a five material volume dataset ('0','1','2','3','4') and working with a two material junction with labels of '1' and '2'. First the material junction of '1' is processed, which is called the primary material. The distance field values at the position of particle p_i, F_i, are interpolated with a B-spline interpolant [80] for all surfaces except label '2', the other material of the junction. Then, for each thread we process the function *max*; the values are summed in threadIdx.x == 0, and the summed value is divided by 2^{m-1} (Eq. 3.10), which is equivalent to $\tilde{f}_i$ in Eq. 3.8. The same procedure is done for material '2' to determine $\tilde{f}_j$ in Eq. 3.8. To prevent the same texture lookups and interpolations from occurring again, however, those texture lookups are stored in an array in CUDA shared memory and the last position is swapped with the primary material. Finally, a similar parallelization strategy is implemented for the computation of ∇*max* (Eq. 3.11) as well.

The GPU implementation of single- and multiple-material mesh extraction provided up to an order of magnitude speed-up over the CPU implementation. However, the Gauss-Seidel update method is an iterative approach and not well



Thread-0	Thread-1	Thread-2	Thread-3
v[0]=tex(0,x,y,z)	v[2]=tex(1,x,y,z)	v[3]=tex(3,x,y,z)	v[4]=tex(4,x,y,z)
v[0]*=g(v-v[0])	v[2]*=g(v-v[1])	v[3]*=g(v-v[3])	v[4]*=g(v-v[4])
v[0]*=g(v-v[4])	v[2]*=g(v-v[0])	v[3]*=g(v-v[1])	v[4]*=g(v-v[3])
v[0]*=g(v-v[3])	v[2]*=g(v-v[4])	v[3]*=g(v-v[0])	v[4]*=g(v-v[1])
v[0]*=g(v-v[1])	v[2]*=g(v-v[3])	v[3]*=g(v-v[4])	v[4]*=g(v-v[0])
val=v[0]+v[2]+v[3]+v[4]			

Fig. 3.12. Example of multithreading a *max*(V). Each material is assigned a different thread for processing Eq. 3.10.

suited for the GPU. The Levenberg-Marquardt update scheme and the projection operator increase the control flow and prevent a one-to-one mapping of the particles to the GPU threads, and the static binning is inflexible with regards to load balancing. Therefore, a more efficient approach is required to fully utilize the GPU.

CHAPTER 4

ENHANCED GPU MESH EXTRACTION

Although speed-ups of up to 44x over the CPU implementation were achieved with the original isosurface GPU implementation as described in Chapter 3, improvements can be made. In particular, the Gauss-Seidel update method is a serial scheme. Although it can be parallelized with a pseudo-red-black Gauss Seidel update, the original Gauss-Seidel update was chosen because each particle advection step attempts to maximize the step-size. This maximization step increases the amount of control flow required, which is antithetical to the very limited branching structure of the GPU. Further, because the binning structure depends on the maximum sizing field value, the bins have unbalanced workloads. Finally, the reprojection step is an iterative search to find the surface and is used frequently while advecting a particle. Because it is an iterative search, it increases the control flow and prevents a one-to-one mapping of particles to threads. Therefore, one goal of this dissertation is to develop an efficient method for particle advection for mesh extraction on the GPU. To increase the efficiency of particle advection for meshing on the GPU, a Barnes-Hut tree code is used to build an acceleration structure and advect the particles [8]. Further, a closest point embedding (CPE) is used to place particles back on the surface after being advected in the tangent plane. Combining these two methods allows for a more efficient particle placement than in [50].

4.1 Barnes Hut Tree Code

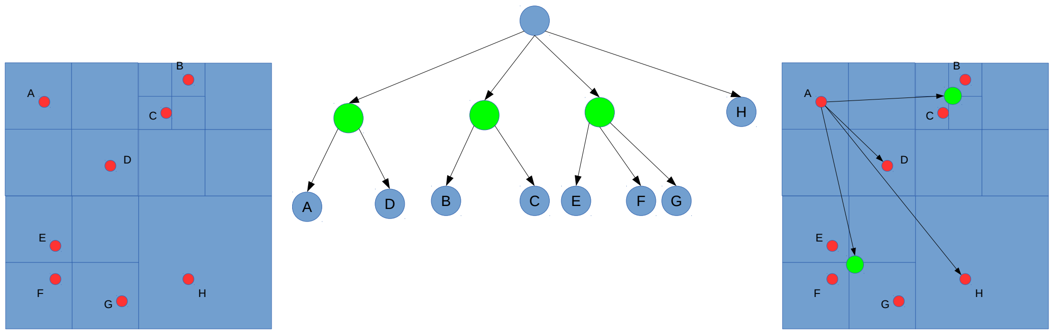
The Barnes-Hut tree code is better suited for particle advection than the previous Gauss-Seidel update method on the GPU. The Barnes-Hut acceleration structure stores all particles in the domain in an octree such that each leaf in the tree has either zero or one particle. Each node links to its children nodes and also contains a representation of its children with a center-of-mass position and the total mass of all its children particles. In practice, the mass of any particle is set to one.

To construct the tree serially, each particle, $\mathbf{p}_i$, is inserted at the root node. Then the particle descends down the tree and at each node updates the center-of-mass and total mass of the node until a leaf is reached. At the leaf, child nodes are created and the center-of-mass of the node is updated.

Once the tree is built, it is used to calculate the force or energy of a particle. For each particle, $\mathbf{p}_i$, the tree is traversed, searching for every other particle, $\mathbf{p}_j$, to calculate the energy or velocity of the particle, $\mathbf{p}_i$. However, during the tree traversal, if the center-of-mass of a node is sufficiently far away from the particle, $\mathbf{p}_i$, then the center-of-mass of the node is used as a single large particle for the energy or velocity computation of particle $\mathbf{p}_i$, and the children of the node are not traversed. The children are not traversed because the farther a particle or group of particles is away from particle $\mathbf{p}_i$, the less effect it has on the energy or velocity of $\mathbf{p}_i$. The children of the node are approximated with the center-of-mass of the node because the cotangent energy function (Eq. 3.1) falls off as the distance between two particles increases. For example, in Fig. 4.1c, the force of particle A is computed by traversing the tree. The particle A directly computes the force from particles D and H, but uses the center-of-mass of particles (B,C) and (E,F,G) instead of traversing the whole tree, which significantly reduces the number of particle-particle energy and velocity computations.

To determine whether a node is "far" away from a particle, a user-defined value, θ, is used. This value is a threshold on the ratio between the size of the node that a particle, $\mathbf{p}_i$, is in, where the size is the edge length defined by the octree level and the distance from the center-of-mass of the node to $\mathbf{p}_i$. In practice, we set θ to 0.75.

The Barnes-Hut tree code eliminates two problems with the Kim *et al.* method [50]. First, the tree code eliminates the need to bin the computational space by the maximum sizing field value. Second, traversing the tree with the Barnes-Hut tree code on the GPU significantly reduces the control flow, which makes the Barnes-Hut tree code a better algorithm for the GPU than the Gauss-Seidel update method. Although all the particles move at the same time with the Barnes-Hut



(a) Particles with octree. (b) Tree representation of (a) (c) Particles and the center-of-mass.

Fig. 4.1. An example of a quadtree decomposition, its tree representation and center-of-mass representation. (a)-(c) are an example of a quadtree built with the Barnes-Hut tree code in two dimensions: (a) has eight particles, $A-H$, with the domain subdivided into an octree, and (b) is the octree from (a), visualized as a tree. The blue nodes that are labeled $A-H$ are leafs, as these are the quads that contain the particles $A-H$ in (a). The nodes colored in green have the center-of-mass of the quadtree's descendants. Finally, (c), is a spatial visualization of the particles with the domain decomposed into quads. The green points are the center-of-mass positions of the nodes, and the particle A has traversed the tree and calculated its energy from the two green nodes and directly from particles D and H.

algorithm, if the step is small enough then the particles will still converge to a good solution.

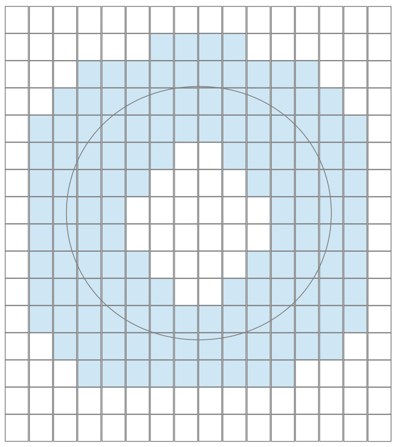
The implementation used in this work to construct and traverse the tree code on the GPU is similar to that of Bédorf *et al.* [8], with two changes. First, the energy and velocity calculations are done with the cotangent energy functions in Section 3.1. The second change is to the velocity calculation. Once the tree code traversal has calculated the velocity for all the particles, $\mathbf{p}_i$, the velocities are multiplied by the sizing field value at the location of the particle, $\mathbf{p}_i$. If the sizing field value is less than zero, i.e., the particles are packed closely because of high curvature, the velocity length is reduced. Likewise, if the sizing field value is large, which indicates an area of low curvature, the velocity length is increased by the sizing field value. By increasing or decreasing the velocity length by the sizing field, the particles will take smaller steps in areas of high curvature and larger steps in areas of low curvature.

A reprojection step (Eq. 3.12) is required to place the particle back on the surface when advecting a particle on an interface (Chapter 3). Unfortunately, this reprojection step is problematic because for each particle, the projection operator is an iterative search to find the surface. Since the projection operator is a search, particles cannot be assigned to individual threads in CUDA because the threads would diverge. This reprojection search limits the performance on the GPU. To overcome this limitation, we convert the distance field into a closest point embedding, which can place the particle directly onto the surface. The preprocessing step of the closest point embedding construction is covered in Section 4.1.1. Once the closest point embedding is constructed, it is used during particle advection to place particles back on the surface, as covered in Section 4.2.1.

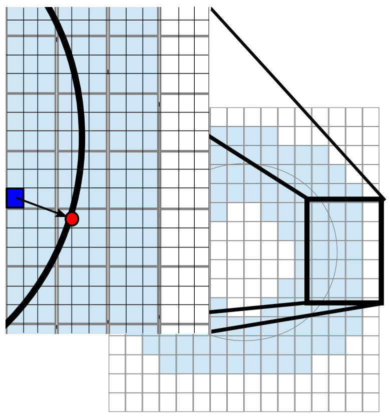
4.1.1 Constructing the Closest Point Embedding

The closest point embedding is reconstructed from the distance field as a preprocess step. A distance field is generated by BioMesh3D, which stores the distance from a cell to the surface, but only for cells close to the surface. Fig. 4.2a is an example of a distance field, where the blue cells are close to the surface, and the white cells are outside the narrow band and do not store a distance to the surface. The closest point embedding stores the location on the surface that is nearest to the cell. Using Fig. 4.2b as an example, the cell at (36,24) is colored blue, and the closest location on the surface to the cell is colored red. The value stored in the distance field at cell (36,24) is 4.3, which is the distance to the surface. However, the value stored in the closest point embedding at the cell (36,24) is (40.2,23.1).

A two-level grid is constructed to store the closest point embedding, similar to Auer *et al.* [2] The grid has two levels, a coarse level and a fine level. The coarse level is a three-dimensional grid with the same dimensions as the distance field, and each cell, corresponding to the narrow band in the distance field, represents a block of subcells for interpolating the closest point position. The fine level is



(a) The coarse grid.



(b) The fine grid.

Fig. 4.2. An example of the closest point embedding using a circle. (a)-(b) are 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. (a) An example surface, a circle embedded in a coarse grid. (b) Part of the fine level of the surface from (a), with spacing $S = 1/3$, and the projection, visualized with an arrow, of the cell in blue, (36,24) to the surface location (the red point) (40.2,23.1).

composed of the subcells of the blocks that are close to the surface and is stored in a one-dimensional array. An example of the coarse grid is in Fig. 4.2a.

4.2 Closest Point Embedding

To construct the closest point embedding, the cells closest to the surface are determined by looking up the corresponding cell value in the distance field. If the value is close to the surface (the blue region in Fig. 4.2b), the cell is processed. For each cell in the one-dimensional fine level, the subcell closest point is computed by projecting the cell position onto the surface using Eq. 3.12. An example is shown in Fig. 4.2b. Once the projection is complete, the closest points are stored contiguously in the fine level array. In practice, to ensure the closest point embedding matches the interfaces generated in BioMesh3D, a Catmull-ROM interpolant is used to project the cells onto the surface [98, 81].

4.2.1 Using the Closest Point Embedding

Once the interfaces are reconstructed into a closest point embedding, a new reprojection step is required to use the closest point embedding 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 (Algorithm 4.1) is used to interpolate the position on the surface [30]. 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 values from the surrounding cells 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, $\mathbf{cp}_i$. It is trivial to expand the interpolation to three dimensions and the three variables, *xyz*, that are required for interpolating the three-dimensional surface position from the closest point embedding.

The Barnes-Hut tree code with the closest point embedding is a more efficient approach to a particle system on the GPU for mesh extraction. By using a tree code, the energy and velocity calculations no longer rely on naive binning, which

Algorithm 4.1 $WENO1d(f_1, f_2, f_3, f_4, x)$

```
    $wp_1 \leftarrow parabola(f_1, f_2, f_3, x)$
    $wp_2 \leftarrow parabola(f_4, f_3, f_2, 1-x)$
    $f \leftarrow (wp_1.x \cdot wp_1.y + wp_2.x \cdot wp_2.y)/(wp_1.x + wp_2.x)$
    return f
```

Algorithm 4.2 $parabola(f_1, f_2, f_3, x)$

```
    $F_x \leftarrow (f_3 - f_1) \cdot 0.5$                          ▷ first derivative
    $F_{xx} \leftarrow f_1 - 2 * f_2 + f_3$                      ▷ second derivative
    $IS \leftarrow F_x * (F_x * F_{xx}) + 1.25 * F_{xx} * F_{xx}$   ▷ smoothness IS
    $IS \leftarrow IS + \epsilon$                              ▷ $\epsilon = 0.000001$
    $IS \leftarrow IS \cdot IS$
    $wp.x = (2 - x)/IS$                            ▷ weight
    $wp.y = f_2 + x \cdot (F_x + 0.5 \cdot x \cdot F_{xx})$            ▷ value at x
    return wp
```

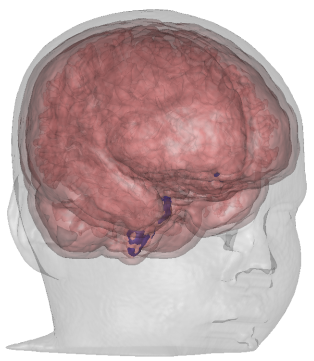
increases workload flexibility on the GPU. The closet point embedding represents the surface on the GPU better than the distance field because it does not require an iterative search to reproject the particles onto the surface. The CPE allows the particles to be mapped to GPU threads in a one-to-one manner, which is more efficient. With this new method, there is up to an order of magnitude performance increase over the multimaterial GPU implementation from Chapter 3.

4.3 Results

In this section, the timing results and the quality of the extracted meshes are discussed. All GPU tests were performed on an Intel Xeon E5-2640 with 32GB of RAM with an Nvidia K20 Tesla card with 5GB of RAM using CUDA 5.0. The CPU tests were performed on an Intel Xeon X5550 with 24GB of RAM. Two datasets, a human *head* and *pig* torso, were used and their initial and final particle count are in Table 4.1. The *head* dataset (Fig. 4.3a) is a four-material volume with a size of (199, 250, 249), and the *pig* dataset is a five-material volume with a size of (136, 136, 136) (Fig. 4.4a). With the new seeding method, the *pig* dataset begins with 286,346 particles, and the *head* dataset begins with 1,283,799 particles. BioMesh3D initial seeding for the *pig* is 151,141 particles, and the *head* dataset begins with 614,344 particles. The *pig* dataset has approximately 350,000 particles when the particle system finishes, and the *head* dataset has approximately 2.1 million particles when the particle system finishes. Both datasets were run with the maximum sizing field value set to 1.0.

TABLE 4.1. The initial and final particle count for the *pig* and *head* datasets, as well as the mean radius ratio of the final mesh.

	CPU			Red-Black Update			Closest Point		
Dataset	Initial #	Final #	Quality	Initial #	Final #	Quality	Initial #	Final #	Quality
pig	151,141	342,231	0.93	151,141	347,906	0.92	286,346	350,498	0.93
head	614,344	1,429,517	0.93	614,344	2,159,347	0.90	1,283,799	2,145,468	0.93


(a) The *head* dataset.

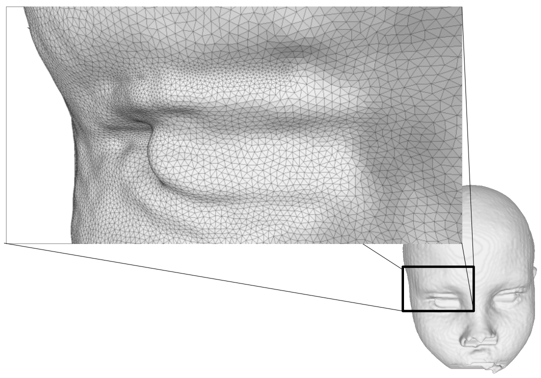
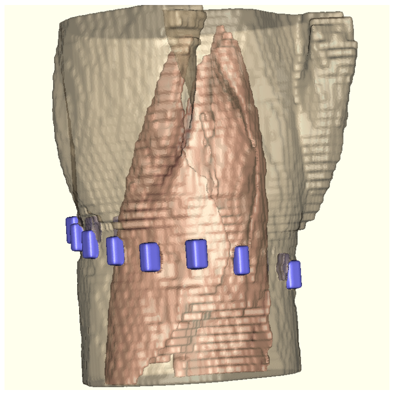

(b) The mesh of the *head* dataset

Fig. 4.3. The *head* datasets: (a) all the material interfaces of the *head* dataset. (b) the triangles of the skin surface.


(a) The *pig* dataset.

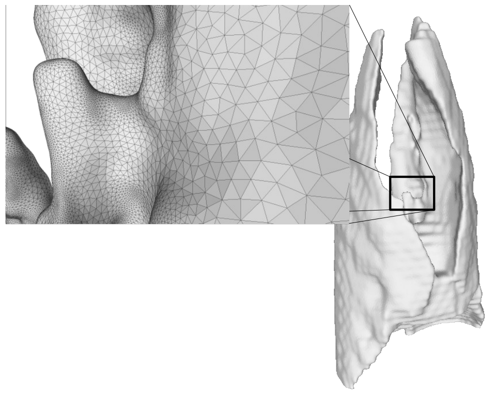

(b) The mesh of the *pig* dataset.

Fig. 4.4. The *pig* dataset: (a) shows all the material interfaces of the *pig* dataset, and (b) shows some of the triangles on the "lung" material of the dataset.

4.3.1 Timing

The timing results of the particle advection for extracting biological multi-material volume data are in Table 4.2. For this timing comparison, the Kim *et al.* GPU implementation was extended to extract multiple materials [50]. The new closest point embedding with the Barnes Hut acceleration structure is 2.8× faster than the red-black update on the GPU for the *pig* and 4.4× faster for the *head* dataset. Further, the closest point with the Barnes-Hut tree code is 10.7× faster and 25.2× than the CPU implementations for the *pig* and *head* datasets, respectively. These are significant performance increases over the previous CPU and GPU implementation. Table 4.3 contains the timing results for preprocessing the multimaterial volume: generating the distance fields and sizing fields as well as the closest point embedding. Further, the timing results for extracting the surface, as examined in Section 3.1, are included. The distance and sizing fields generated with BioMesh3D are shared with the CPU, the red-black, and closest point implementations. Fig. 4.5 combines the timing results for the distance field, sizing field, and surface extraction from Tables 4.2 and 4.3 into a normalized stacked

TABLE 4.2. A comparison of time (in seconds) to complete particle advection for the closest point embedding with the CPU, the red-black implementation (RBGS), and the Barnes Hut tree code (BH). The datasets are the *pig* torso and human *head* volumes.

	Time (secs)			Speed-up		
Dataset	CPU	RBGS	BH	CPU vs RBGS	BH vs CPU	BH vs RBGS
pig	5,056	1,312	472	3.9x	10.7x	2.8x
head	42,725	7,445	1,694	5.7x	25.2x	4.4x

TABLE 4.3. Timing results for the *GenerateDistanceVolumes*, *GenerateSizingFields*, closest point grid generation, and *SurfaceExtraction* (Section 3.1) for the *pig* and *head* datasets. All results are in seconds.

	Preprocessing			Postprocessing	
Dataset	Distance Volume	Sizing Field	Closest Point	Extract	Total Time
pig	60	723	6	17	800
head	1,135	3,129	35	70	4,334

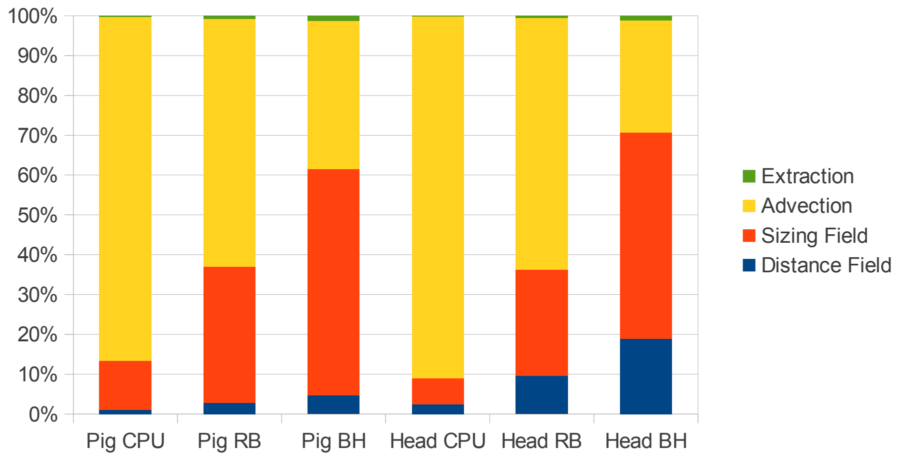


Fig. 4.5. A normalized chart of the full timing results for the *pig* and *head* datasets from Tables 4.2 and 4.3. Each bar is the normalized time to extract the mesh (generating the distance field, generating the sizing field, advecting the particles, and extracting the mesh) using the CPU, the red-black implementation, and the Barnes-Hut tree code.

chart. Each bar is stacked with distance field, sizing field, particle advection, and mesh extraction timing results, from bottom to top, respectively. The closest point grid generation constitutes less than 1% of the preprocess time and has been omitted from the chart.

With the *pig* dataset, the particle advection section takes 86% of the total time to extract the mesh on the CPU, a significant portion of the total time to extract the mesh. With the red-black implementation, the advection takes 63% of the total time. Although an improvement over the CPU, the particle advection still requires 1.8x more time than generating the sizing field, the segment that takes the second most amount of time. Finally, the closest point embedding with Barnes-Hut tree code is 37% of the total time to extract the mesh, which is a smaller portion of the total time than the sizing field generation, which is 57% of the total time. This increase in performance is illustrated in Fig. 4.5. The three bars on the left are the timing results for *pig* dataset for the CPU, the red-black, and closest point implementations. These normalized graphs show that particle advection takes a significant portion of the time on the CPU. With the closest point embedding, though, particle advection takes less time than the sizing field generation. With the Barnes-Hut tree code, particle advection is no longer the bottleneck for extracting conformal meshes from multimaterial data.

With the *head* dataset, the particle advection takes 91% of the total time to extract the mesh on the CPU. With the red-black implementation, the advection takes 63% of the total time, but still needs 2.4x more time than the segment that takes the second most time, the sizing field generation. Finally, the closest point embedding with Barnes-Hut tree code is 28% of the total time and is almost twice as fast the sizing field generation, which is 54% of the total time. The *head* dataset timing is also in Fig. 4.5, where the three bars on the right are the timing results for the *head* dataset for the CPU, the red-black, and closest point implementations. Like the *pig* dataset, these normalized bars of the *head* dataset illustrate that particle advection takes a significant portion of the time on the CPU. With the closest point embedding, though, particle advection takes less time than the sizing field generation. Again, as with the *pig* dataset, particle advection is no longer the bottleneck in the particle-based multimaterial meshing pipeline with the closest point embedding and the Barnes-Hut tree code.

The Barnes-Hut tree code with closest point embedding is up to 4.4x faster than the fastest known GPU particle advection for multimaterial mesh extraction. Further, it is up to 25.2x faster than BioMesh3D. This new technique removes the largest bottleneck of the multimaterial, conformal mesh extraction pipeline and would facilitate the adoption of the particle system pipeline in the biomedical community [94].

4.3.2 Quality

To measure the quality of the mesh, the multimaterial triangular mesh is constructed using the particle locations. With the triangular mesh, the inscribed/-circumscribed radius ratio of every triangle in the mesh is calculated and averaged over the entire mesh. A numerical value of 0.90 or greater for the radius ratio is considered to be an adequate triangular mesh, whereas a value over 0.92 is considered to be a high-quality triangular mesh and a good starting point for generating a good tetrahedral mesh. The *head* and the *pig* dataset have an average inscribed/circumscribed radius ratio of 0.93. This ratio is better than the ratios of the meshes from the red-black update scheme. Further, both the *pig* and *head* dataset average radius ratio are above 0.92 and considered to be high-quality triangular meshes.

By eliminating the iterative search for the surface with the closest point embedding and using a flexible acceleration scheme, the Barnes-Hut tree code, the new particle update scheme works better on the GPU. This new technique for particle advection for mesh extraction is faster than the previous GPU implementation and generates high-quality triangular meshes.

4.4 Discussion

In this chapter we have presented a new isosurface extraction algorithm with the closest point embedding. This new technique, coupled with a GPU Barnes-Hut tree code, is used for curvature-adaptive, multimaterial mesh extraction from labeled volume data. The closest point embedding is a faster method for the GPU because the reprojection step is no longer an iterative search. Each particle can be assigned to a thread without the need for an iterative search for the surface in the reprojection step. Further, the Barnes-Hut tree code is better suited for particle advection on the GPU because instead of maximizing the step for each particle, small velocity steps are taken to reach an optimal particle configuration on the surface. These small velocity steps remove much of the control flow that hindered the performance in previous GPU implementations of particle systems for surface extraction. The closest point embedding with Barnes-Hut tree code is faster than any known particle system for multimaterial mesh extraction. The speed-up is transformative for biomedical work such as Electrical Impedance Tomography (*EIT*) Imaging of the lung [94].

CHAPTER 5

SURFACE FLOW VISUALIZATION USING THE CLOSEST POINT EMBEDDING

In the previous chapter, the surface representation from the closest point method (CPM), the closest point embedding, was used as an embedding surface to project particles back onto the surface. The CPM was originally developed for solving partial differential equations on surfaces with normal three-dimensional stencils. The closest point embedding provides a functionality that gives us an opportunity to solve an interesting surface flow problem: surface flow visualization on an arbitrary surface.

Surface flow visualization techniques can be classified into two categories: parameterization methods and image space techniques. The state of the art in parameterized methods is Flow Charts, which decomposes the surface into patches and packs those patches into a two-dimensional atlas [65]. The parametric method was chosen for Flow Charts because image space techniques, such as Image Based Flow Visualization (IBFV) [111], are limited by the image space parameterization, and any self-occluded surface will be incoherent. Further, user movements can create "popping" when the surface is rotated because the image space parameterization is not fully consistent between frames. However, parameterizing a surface is difficult, especially for complex geometry. For instance, Flow Charts requires a lengthy preprocess step that prevents the parameterized surface from being generated at an interactive rate.

This chapter describes a new method for surface flow visualization to solve the problem of artifacts with image-space based techniques without the difficulties of parameterizing the surface by using the closest point embedding with particle-based flow techniques such as GPUFLIC [63].

5.1 Closest Point Embedding Construction

In Chapter 4, the closest point embedding was constructed by projecting grid cells onto the surface using a distance field (Section 4.1.1). Since surface flow usually begins with a mesh, it must be converted to the closest point embedding.

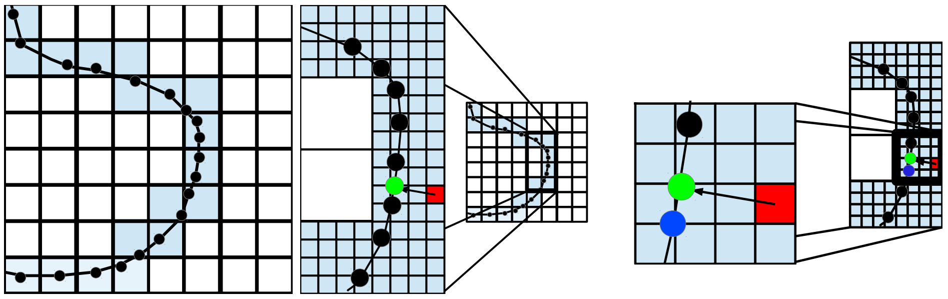
The closest point embedding accomplishes two objectives. First, the closest point grid is used to project UFLIC particles back onto the surface (Section 5.1.1). Second, the closest point embedding is used to generate a refined grid and a neighborhood index (Section 5.2.1). This neighborhood index is used to run high-pass filtering and antialiasing 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 [43, 83]. Constructing the closest point embedding is covered in Section 5.1.1. Once the closest point embedding is constructed, it is used during particle advection to place particles back on the surface, which is covered in Section 5.1.2.

5.1.1 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. Fig. 5.1a 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 Fig. 5.1b 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)*.

A two-level grid is constructed to store the closest point embedding, similar to Auer *et al.* [2]. 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 subcells for interpolating the closest point position. The fine level is composed of the subcells



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

Fig. 5.1. An example of the coarse and fine levels of the closest point embedding. (a)-(c) are two-dimensional examples of the closest point embedding. For all figures, the cells close to the surface are colored blue, and cells far away from the surface are colored white. (a) is an example surface, a curve embedded in a coarse grid. (b) displays part of the fine level of the surface from (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)* , the projection is visualized with an arrow, and the surface location (the green point) is at *(21.3,14.8)*. Finally, (c) focuses on the fine grid cell (from (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.

of the coarse grid cells and is stored in a one-dimensional array. This two-level grid saves memory by refining only the coarse grid where the cells are close to the surface.

Construction of the closest point embedding is shown in Algorithm 5.1. 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." Fig. 5.1a is an example of a one-dimensional curve embedded into a coarse two-dimensional grid. The coarse grid cells colored blue are "on surface," whereas white cells are considered far away. Next, each coarse grid cell that is "on surface" is subdivided to create the fine grid cells. Once all the fine grid cells are determined, the closest point on the triangular surface is computed and stored in a one-dimensional fine grid array: one for each coarse grid cell.

Algorithm 5.1 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 $FG$ [96])
            $d_{new} = ||fpt - FG||$
            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
```

To construct the fine grid cells, each coarse cell that is "on surface" is subdivided into fine cells. Fig. 5.1b 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. 5.1c. 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 faces adjacent 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 and the two other vertices are in a coordinate plane. This transformation converts 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 [96]. Fig. 5.2 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 process 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.

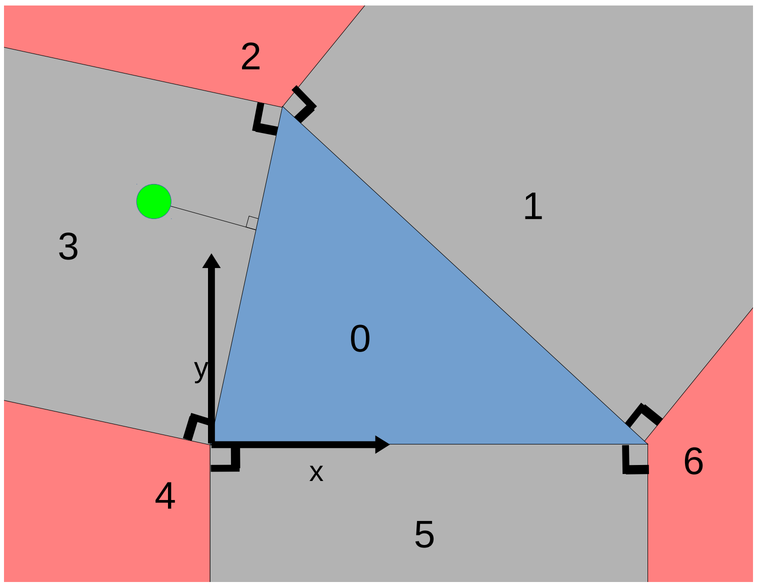


Fig. 5.2. An example of a triangle face (in blue) projected into a coordinate plane and the seven different regions numbered. The green vertex is a grid vertex projected into the two-dimensional plane and is in region 3.

5.1.2 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 onto the surface after the advection method. To place a particle back onto the surface with closest point embedding, a WENO4 interpolant (Algorithms 5.2 and 5.3) is used to interpolate the position on the surface [30]. 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, $\mathbf{cp}_i$.

5.2 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 adapt UFLIC to the closest point embedding.

Unsteady Flow Line Integral Convolution (UFLIC) is a technique to visualize two-dimensional unsteady flow [99]. 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.

Algorithm 5.2 *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. 5.3
    $wp_2 \leftarrow parabola(f_4, f_3, f_2, 1 - x)$
    $f \leftarrow (wp_1.x \cdot wp_1.y + wp_2.x \cdot wp_2.y)/(wp_1.x + wp_2.x)$
    return f
```

Algorithm 5.3 *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
    $IS \leftarrow IS + \epsilon$                                       ▷ $\epsilon = 0.000001$
    $IS \leftarrow IS \cdot IS$
    $wp.x = (2 - x)/IS$                                                 ▷ weight
    $wp.y = f_2 + x \cdot (F_x + 0.5 \cdot x \cdot F_{xx})$             ▷ value at x
    return wp
```

5.2.1 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 Section 5.2 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 antialiasing the three-dimensional pathlines.

To construct the sparsely stored refined grid, the closest point grid from Section 5.2 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 Fig. 5.3, 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 antialiasing 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 (Section 5.1.2). Then, the index of the ncp_i is computed, idx_{ncp}, and stored in the neighboring index array. By storing the neighboring indices, the

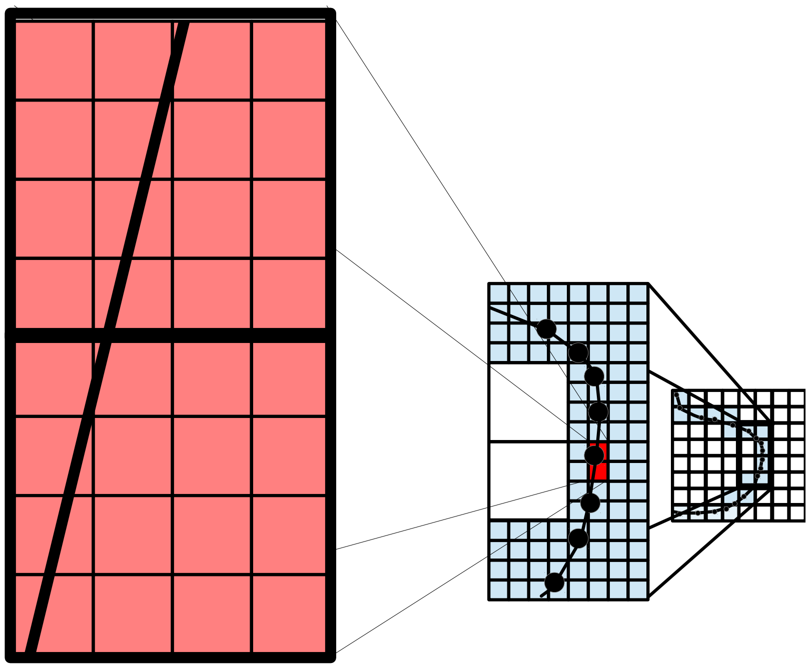


Fig. 5.3. To construct the sparsely stored refined grid, the closest point embedding is subdivided. Using the original two-dimensional closest point embedding example from Fig. 5.1b, the fine grid is subdivided and two grid cells are each subdivided into eight refined grid cells, which are shown in red.

Laplacian filter can be applied directly on the refined grid and the antialiasing of the pathline is sped up.

For example, in Fig. 5.4, 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.

5.2.2 Unsteady Flow Line Integral Convolution

To adapt unsteady flow line integral convolution (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 advect the particle

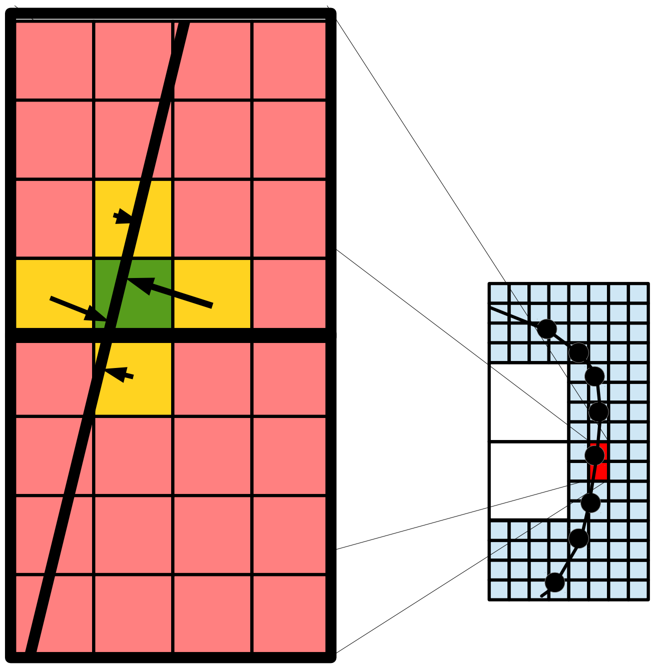


Fig. 5.4. Continuing with the two-dimensional fine grid example from Fig. 5.3, a single refined grid cell is highlighted in green, with its four neighbors colored yellow.

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 Fig. 5.5. In Fig. 5.5a, the pathline ends off the surface, i.e., in a white cell. The pathline length is cut in half (Fig. 5.5b), but again the pathline terminates off the surface in a white cell. The pathline is halved a third time (Fig. 5.5c), 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 (Section 5.1.2) and becomes the new starting point, as in Fig. 5.5d.

To draw the piecewise pathline, a three-dimensional Bresenham algorithm [13] (Algorithm 5.4) is used and adapted for antialiasing. To antialias the line, a low-pass

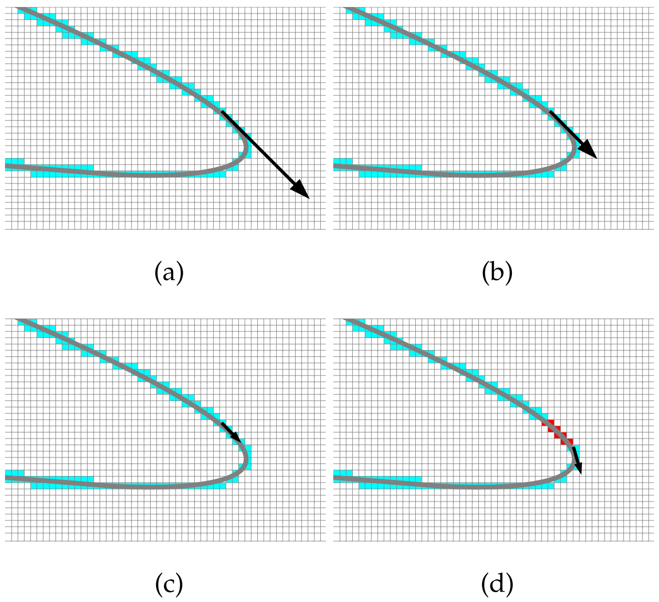


Fig. 5.5. Two-dimensional examples of the pathlines being halved until the particle is on the surface. In (a), the original pathline does not end in a cell near the surface (cells colored blue). Therefore, in (b) the length is cut in half, but again the pathline does not end in a cell near the surface, and in (c) the pathline is reduced again. The pathline now terminates on a cell close to the surface, and a pathline is drawn, shown in red in (d). A new pathline is started in (d) where the previous pathline ended using the previous pathline's length. Drawing pathlines in this manner is repeated until the original pathline length is drawn.

Algorithm 5.4 Antialiased Three-Dimensional Line Algorithm()

```
        ▷ Input: Begin point $begin$, end point $end$ and scalar val, $val$. Output:
three-dimensional, antialiased line on the refined grid
int3 $p1 = floor(begin)$
int3 $p2 = floor(end)$
$p \leftarrow float3(p1)$
$d \leftarrow float3(p2 - p1)$
$N \leftarrow max(abs(d))$
$s \leftarrow d/N$
**for** $i := 0 \rightarrow N$ **do**
    **if** $s.z = 1$ **then**
        Update neighbors in the xy-plane by $val \div 8$.
    **else**
        **if** $s.y = 1$ **then**
            Update neighbors in the xz-plane by $val \div 8$.
        **else**
            Update neighbors in the yz-plane by $val \div 8$.
        **end if**
    **end if**
**end for**
```

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 is updated in a similar fashion, respectively.

5.2.3 UFLIC With the Closest Point Embedding

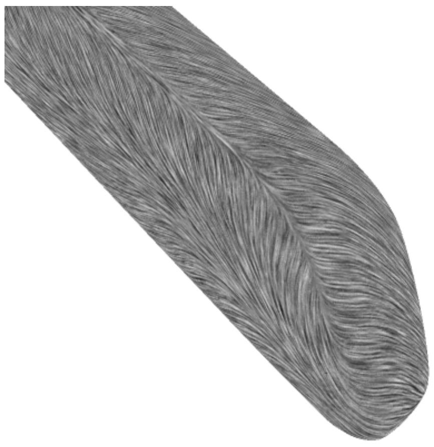
To run UFLIC on the closest point embedding, initially a white noise refined grid is created. Given closest point and velocity grids, the refined grid is constructed as in Section 5.2.1. Once the refined grid is constructed, each refined grid cell is seeded with a particle, and the particle is projected onto the surface using the WENO4 from Section 5.1.2. 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 Section 5.2.2.

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 [99]. A three-dimensional 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.

5.3 Results

To test this new method, three datasets are used: the ICE train, the F6 plane, and the cylinder combustion datasets (Figs. 5.6, 5.7, and 5.8, respectively). An important goal is that the closest point embedding has comparable results to Flow Charts [65], so each dataset has a figure using Flow Charts for comparison purposes.

The ICE train (Fig. 5.6) 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 Fig. 5.6a. Shear stress is shown on the airliner (F6) dataset, which is in Fig. 5.7a. The combustion dataset (Fig. 5.8) is a complex combustion cylinder with input and exhaust pipes as well as valves inside the combustion chamber. The


(a) The Closest Point ICE Train

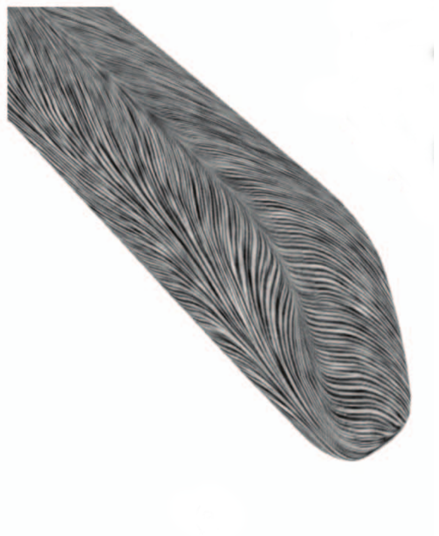
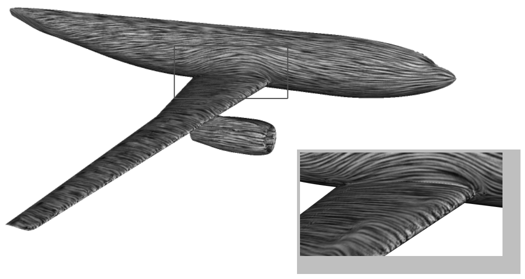

(b) The Flow Charts ICE Train

Fig. 5.6. The ICE train visualized with UFLIC with (a) closest point embedding and using (b) Flow Charts.


(a) The Closest Point F6

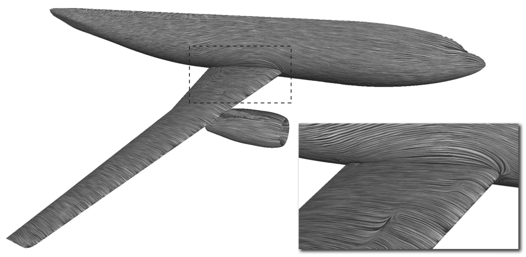
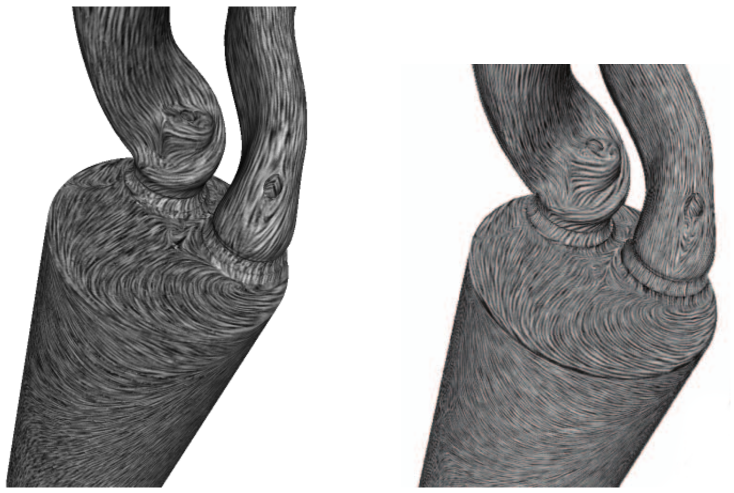

(b) The Flow Charts F6

Fig. 5.7. The airliner (F6) dataset visualized with UFLIC and (a) closest point embedding and (b) using Flow Charts.



(a) Closest Point Cylinder (b) Flow Charts Cylinder

Fig. 5.8. Engine cylinder visualizations. The cylinders in (a) and (b) use UFLIC with the closest point embedding and Flow Charts, respectively, for visualizing flow in a combustion cylinder.

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 Fig. 5.8a.

The timing results and the dimensions of the closest point grid and refined grid for the datasets are in Table 5.1 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 are usually helper arrays to compact other arrays in Thrust. Permanent data are added at the head of the memory pool, whereas temporary data are added to the tail of the memory pool. This way, permanent

TABLE 5.1. 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	Refined	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	$(144 \times 222 \times 472\ )$	$(432 \times 666 \times 1416)$

arrays are not interleaved with temporary arrays, and the temporary data can be pushed and popped 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.

5.4 Discussion

In this chapter, a new method for surface flow visualization using the closest point embedding was introduced. 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. Our technique 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 lengthy preprocess step such as Flow Charts.

CHAPTER 6

CLOSEST POINT SPARSE OCTREE AND UNSTEADY SURFACE FLOW

As datasets continue to grow in size and complexity, surface visualization techniques need to scale to address this challenge. In Chapter 5, surface flow visualization with the closest point embedding (SFCPE) achieved near-interactive rates for unsteady flow line integral convolution. The SFCPE uses a two-level coarse grid/refined subgrid to represent an embedded surface. The coarse grid is the closest point grid, whereas the refined subgrid is used to visualize the surface flow. The coarse grid does not scale well as the grid resolution increases because it is a three-dimensional dense grid. To visualize increasingly large and complex surface flows, we introduce the closest point sparse octree (CPSO) to represent an embedded surface. By using a sparse octree, regions that are not near the embedded surface are skipped, which saves memory over a dense grid and scales to higher grid resolutions. The CPSO is, to our knowledge, the first sparse octree for the closest point method.

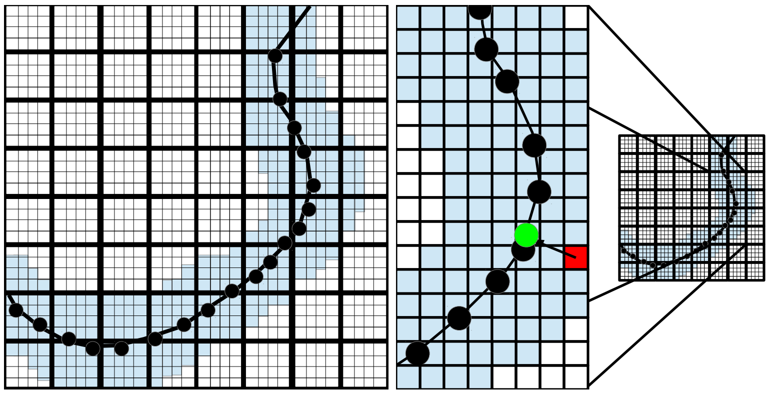
In addition to the CPSO, an unsteady flow visualization technique is implemented for the closest point method. In Chapter 5, a particle reprojection method was used to perform unsteady flow line integral convolution, UFLIC, on the closest point embedding [52]. Instead of using a projection step, UFLIC is adapted for the closest point method, which solves partial differential equations on embedded surfaces [93]. By extending the velocity field and surface values into the embedding grid, the particle advection portion of UFLIC can be performed in three dimensions without the particle reprojection step. Further, standard three-dimensional operators, such as the Laplacian operators, can be applied directly to the surface instead of using intrinsic operators, which simplifies the implementation.

6.1 Embedding the Surface

To maintain as much flexibility as possible, we implement the sparse voxelization octree approach similar to Baert *et al.* [3] 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.

6.1.1 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. 6.1a is a two-dimensional grid, where the blue cells



(a) A piecewise curve embedded into a grid.

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

Fig. 6.1. 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. An example surface, a curve embedded in a $24x24x24$ 2D grid is in (a) . In (b), part of the grid from (a) is displayed 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)*.

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. 6.1b 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 shown in Algorithm 6.1. 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.

Then, for each triangle in the subgrid, an axis-aligned bounding box (AABO) is determined. This AABO is expanded by a user-defined offset. In practice, the offset is set to 3. Then, for each grid cell in the AABO, 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 (AABO), 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

Algorithm 6.1 BuildClosestPointOctree() Input: Triangular Mesh, *TM* with velocity field *VM*, Grid *G* Output: Sparse closest point octrees

```
for all subgrid, $SG \in G$ do
    Initialize all cell counts $cnt \in SG$ to 0
    for all Triangles $t_j$ in grid $G$ do
        $AABB_j \leftarrow$ axis-aligned bounding box of $t_j$
        $AABB_j \leftarrow$ expand in each direction by offset $o$
        for all cells $c_k \in AABB_j$ do
            $cnt_k \leftarrow cnt_k + 1$
        end for
    end for
                                        ▷ copy the index if cnt is greater than 0
    $tri\_lookup\_idx = exclusivescan(cnt)$
    $tri\_idx \leftarrow 0$
    $tri\_cnt \leftarrow 0$
    for all triangles $t_j \in SG$ do
        $AABB_j \leftarrow$ axis-aligned bounding box of $t_j$
        $AABB_j \leftarrow$ expand in each direction by offset $o$
        for all cells $c_k \in AABB_j$ do
            $tri\_idx_{tri\_lookup_k} \leftarrow tri\_cnt_k$
            $tri\_cnt_k \leftarrow tri\_cnt_k + 1$
        end for
    end for
    for all cells $c_i \in SG$ do
        if $tri\_cnt_i > 0$ then
            $dist = 1e6$
            for all triangles $t_j \in tri\_idx_i$ do
                $cpm \leftarrow ClosestPoint(c_i)$
                if ( then$|(cpm - c_i)| < dist$)
                    $dist \leftarrow |(cpm - c_i)|$
                    $cpm_i \leftarrow cpm$
                end if
            end for
        end if
    end for

end for
```

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 [96].

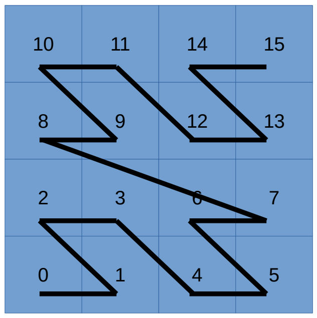
To ensure scalability of the closest point construction, the grid is subdivided into subgrids depending on the amount of memory on the GPU. The number of partitions required is determined by the amount of memory needed to store a Morton code (8 bytes), a closest point (12 bytes), and a velocity vector (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.

6.1.2 Morton Order

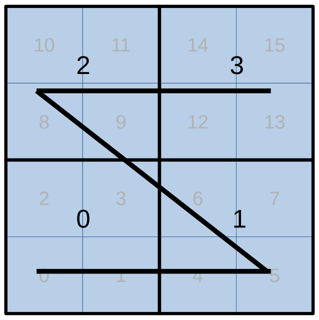
Morton order, or z-order (Fig. 6.2), is a multidimensional to one-dimensional mapping that maintains locality. It is a hierarchical ordering such that the Morton order for a high level of the tree (Fig. 6.2a) is congruent to the Morton order of a lower level (Fig. 6.2b) of the octree. The purpose of 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 6.2). To interleave the bits, for each bit b at position i in a grid cell coordinate c, a mask is created ($1 << i$) and *and*ed to that coordinate. Then, it is bitshifted by twice the bit position i and *or*ed to the output. 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 z-order.

6.1.3 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 in Section 6.1.1, 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



(a) Highest level Morton order.



(b) Level below the highest level Morton order.

Fig. 6.2. A two-dimensional example of Morton order and its hierarchy: (a) is the highest level Morton order, and (b) is a coarser Morton order.

Algorithm 6.2 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**
```

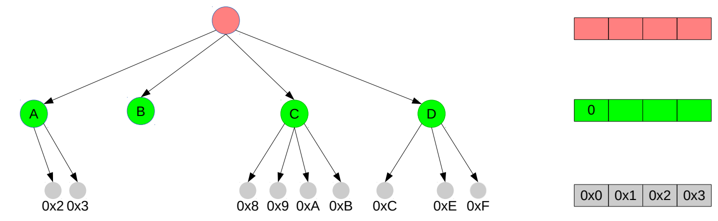
is recorded. The lowest level queue is reused for the next set of leaf nodes. This parent-child relationship recording is recursively done for each queue of the tree, until the tree is completed. Note, if a queue is filled with empty keys, then the queue can be skipped and a key inserted at the parent 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. 6.3. At the highest level of the octree, Morton keys $0x0$ to $0x2$, along with empty keys $0x2$ and $0x3$, 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, for 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, 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. 6.3 is an example with the queue stopped after the parent-child relationship is recorded for the green level A.

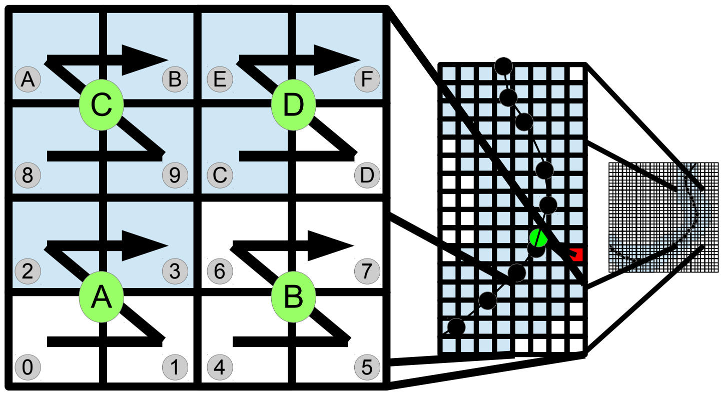
6.1.4 Using the Closest Point Octree

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 *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 storing its index in a neighborhood lookup.



(a) Sparse octree construction



(b) Sparse octree example

Fig. 6.3. Continuing with the embedded piecewise curve example from Fig. 6.1, 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 $0x0$ and $0x1$ are leaf nodes that exist in the closest point grid. The nodes $0x2$ and $0x3$ are empty key.

6.2 Flow Visualization With the Closest Point Method

To demonstrate the effectiveness of the closest point method for flow visualization, we apply 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.

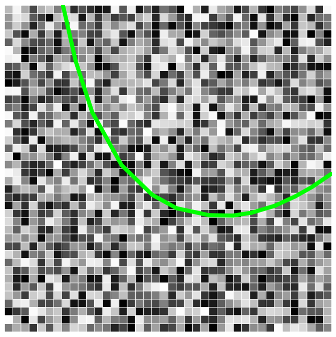
UFLIC is a technique to visualize unsteady flow [99]. In this scheme, particles are released from the center of every pixel and are advected forward, depositing their scalar value along the pathline. Once the advection and depositing are completed, the accumulated values are normalized, filtered, and jittered, creating the flow visualization.

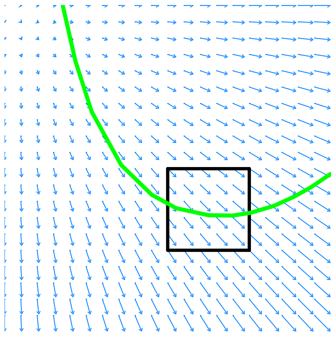
6.2.1 UFLIC

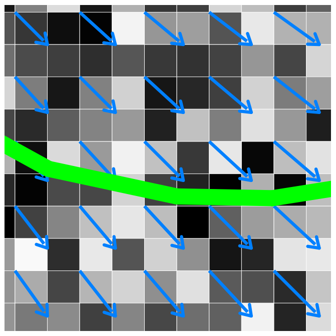
The closest point sparse octree is used to produce and visualize the UFLIC on the surface. Initially, given the closest point and velocity grid, a UFLIC sparse grid of the same size is filled with random noise, similar to how a two-dimensional UFLIC is initialized. Once the noise grid is constructed, 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.

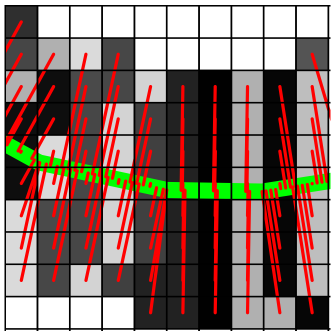
Once the white noise has been extended into the extension grid, for each cell, a point is placed at the center of the cell and stores the value of the initial grid cell. As the particles are advected through the grid, their initial values are accumulated in the UFLIC grid using a three-dimensional Bresenham line drawing algorithm. Once the advection process is complete, the field is normalized, sharpened with a three-dimensional laplacian operator, and jittered. To visualize the surface, a parametric CPU raycaster is implemented [92].

An example of the extension and application of UFLIC is shown in Fig. 6.4. Fig. 6.4a and 6.4b are a 40×40 noise grid and a 20×20 velocity field, respectively. The velocity field is down sampled to reduce visual clutter. Fig. 6.4c shows the zoomed-in region of Fig. 6.4b combined with the noise of 6.4a. Fig. 6.4d extends the surface values into the extension grid, where the closest point to a grid cell is


(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.

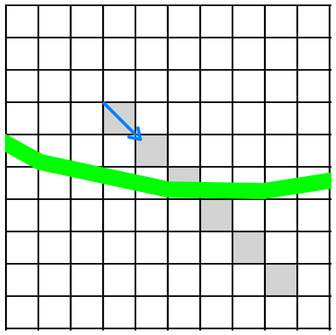

(e) Applying UFLIC.

Fig. 6.4. A two-dimensional example of the UFLIC with a one-dimensional embedded curve. A 40×40 noise grid is in (a) and a vector field is in (b), but 20×20 to reduce visual clutter. A zoomed-in portion for (b) is in (c) and (d). The extension phase of the closest point method is in (c) . The values have been interpolated into the UFLIC grid. Finally, a single example is given in (e), where a value that was interpolated from the surface is then deposited back onto the surface with UFLIC.

shown with a red line. Finally, UFLIC is applied to the two-dimensional extended grid in Fig. 6.4e, 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.

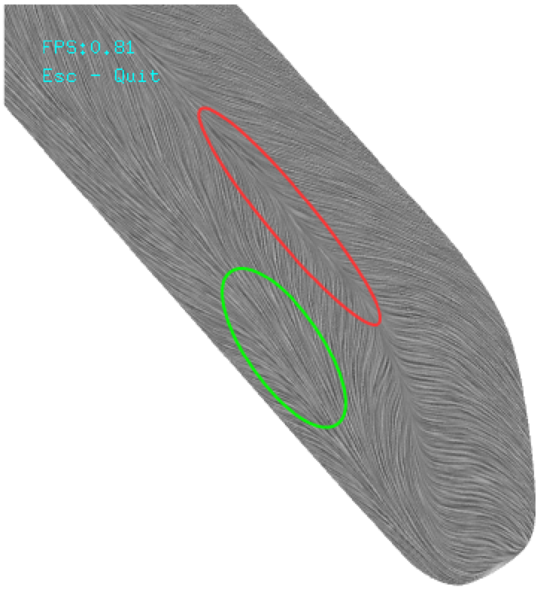
After the particles are advected through the surface, the UFLIC grid is normalized and a standard three-dimensional laplacian filter is applied to all the cells. Then, the grid is clamped and jittered to prepare for the next iteration of UFLIC.

6.3 Results and Discussion

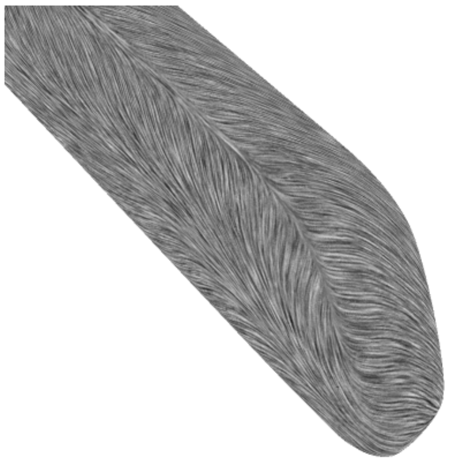
To validate the UFLIC on a surface using the closest point method, a visual comparison is performed between our technique and two previous unsteady surface flow visualization methods: Flow Charts [65] and SFCPE [52]. Then, to demonstrate the CPSO performs and scales well, three datasets are used with varying grid sizes.

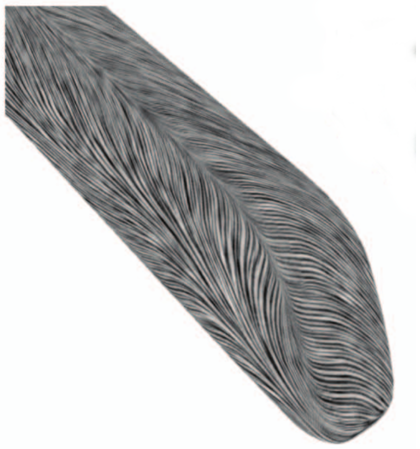
6.3.1 Validation

To validate UFLIC with the CPSO, two datasets are used: the ICE train and the F6 aircraft datasets (Fig. 6.5 and 6.6, respectively). An important goal is that the CPSO has comparable results to previous parametric unsteady flow surface visualization techniques: Flow Charts [65] and the SFCPE [52]. A figure is provided for each dataset using Flow Charts and SFCPE, as well as the CPSO, for comparison purposes. Both the ICE train and the F6 aircraft 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 aircraft was chosen because it is visually similar to the Flow Charts and SFCPE using 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.



(a) The CPSO ICE Train

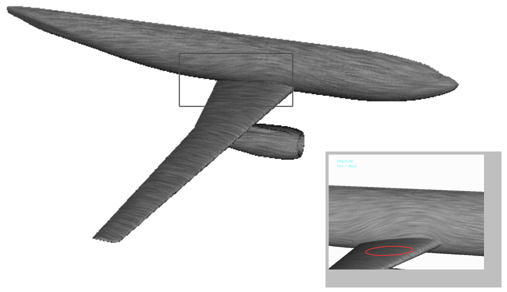


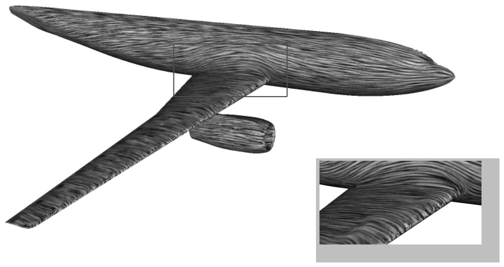


(b) The SFCPE ICE Train

(c) The Flow Charts ICE Train

Fig. 6.5. The ICE train visualized with UFLIC with the CPSO (Fig. (a)), SFCPE (Fig. (b)), and using Flow Charts (Fig. (c)).


(a) The CPSO Aircraft


(b) The CPE Aircraft

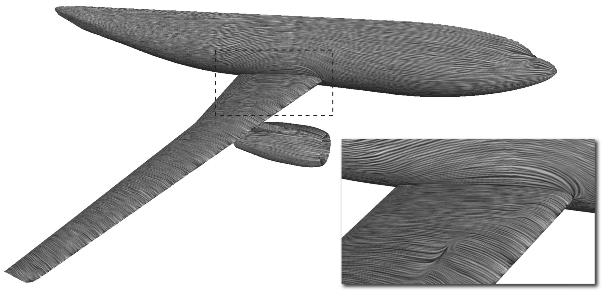

(c) The Flow Charts Aircraft

Fig. 6.6. The F6 aircraft dataset visualized with UFLIC and the CPSO (Fig. (a)) the SFCPE (Fig. (b)) and using Flow Charts in Fig. (c).

The ICE train 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 Fig. 6.5a. Shear stress is shown on the F6 aircraft dataset, which is in Fig. 6.6a.

When generated with Flow Charts, the SFCPE, and the CPSO, both datasets are visually similar. For the ICE train in Fig. 6.5, the separation (highlighted by the red circle) and the attachment (highlighted by the green circle) flow patterns can be seen in all three procedures. With the F6 dataset in Fig. 6.6, the shear stress from the wind (highlighted with a red circle) can been seen in all three implementations as well.

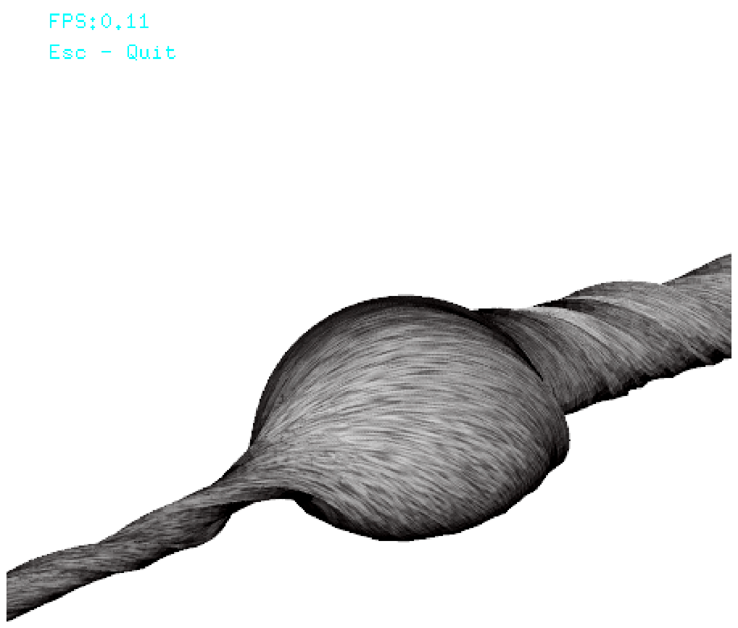
6.3.2 Timing and Scaling Results

To demonstrate the effectiveness of the CPSO, 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 (Section 6.3.1), the ICE train and the F6 aircraft 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. 6.7), 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 6.1. 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 CPSO 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 (Section 6.1.4). 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.



(a) The CPSO delta wing vortex bubble

Fig. 6.7. The delta wing vortex bubble dataset is a stream surface off of a delta wing. It is a complex surface that flows around itself. Fig. (a) shows the surface of the delta wing vortex bubble.

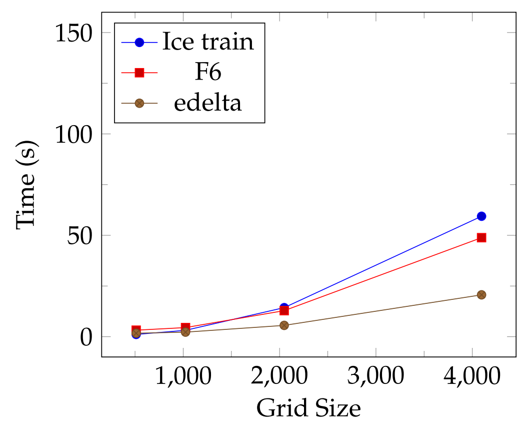
TABLE 6.1. The timing results (in seconds) and the increase in time from the previous grid size for the construction of the CPSO and applying UFLIC as well as dimensions for the datasets are listed. Further, the number of sparse voxels and the increase from the previous grid size voxel count are listed in the last two columns.

	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 Aircraft	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

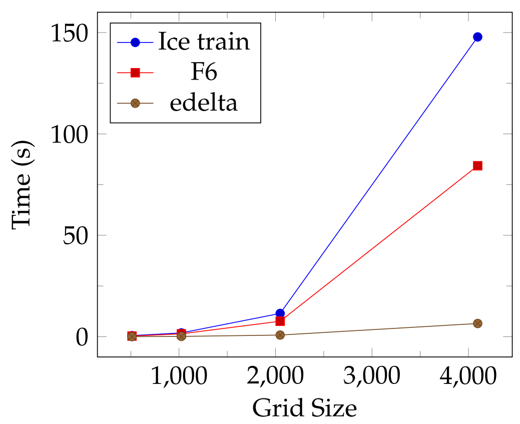
For the CPSO construction (Fig. 6.8a), the amount of time it takes to construct the sparse octree scales at similar rate as the number of sparse voxels (Fig. 6.8c) rather than increasing exponentially with the dense grid size. Table 6.1 includes the timing results for building the CPSO and applying UFLIC 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 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. This tree traversal is the cause for the UFLIC performance not scaling linearly with the number of voxels. Table 6.2 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.

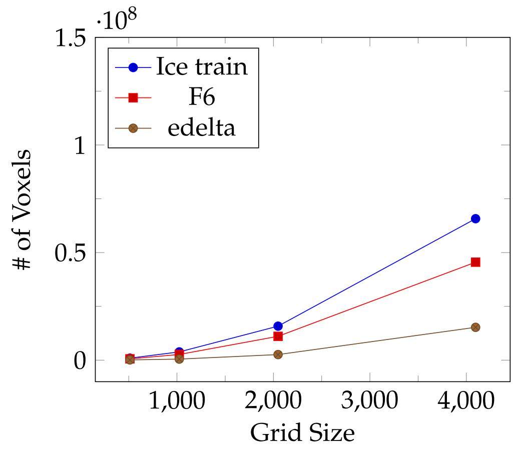
Similar to the linear scaling of the CPSO construction and voxel count of the ICE train, the F6 aircraft dataset's CPSO construction time increases linearly, as



(a) Sparse Octree Construction Timing



(b) UFLIC Timing



(c) Voxelization Count

Fig. 6.8. Three graphs from data in Table 6.1. Fig. (a) and (b) are graphs of the timing results of the CPSO construction and applying UFLIC, respectively. Fig. (c) is a chart of the number of voxels that are generated from the sparse octree.

TABLE 6.2. 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.

	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

does the voxel count. The voxel counts for the F6 aircraft are 611,854, 2,647,537, 11,078,157, and 45,526,355 for grid sizes 512^3, 1024^3, 2048^3, and 4096^3, respectively. Further, the construction times for the CPSO (in seconds) are 3.22, 4.5, 12.85, and 48.84 for grid sizes 512^3, 1024^3, 2048^3, and 4096^3, respectively. The increase in the voxel count from grid size 512^3 to 1024^3 is 4.3x, the increase from grid size 1024^3 to 2048^3 is 4.2x, and the increase in the voxel count from grid size 2048^3 to 4096^3 is 4.1x. Similarly, the increase in CPSO time as the grid size increases is linear. Increasing the grid size from 512^3 to 1024^3 increases the CPSO construction time by 1.4x, whereas expanding the grid size from 1024^3 to 2048^3 increases the construction time by 2.9x. Finally, increasing the grid size from 2048^3 to 4096^3 is an increase in the construction time of 3.8x. Like the ICE train dataset, applying UFLIC is a nonlinear increase in time. The increase in time for grid size 512^3 to 1024^3 is 4.6x, the increase in time for grid size 2048^3 is 5.4x, and the increase in time for grid size 4096^3 is 11.0x. The nonlinear increase in UFLIC runtime is because the neighborhood lookup is not used.

Finally, the scaling of the CPSO and voxel count of the delta wing vortex bubble dataset are similar to ICE train and F6 aircraft datasets. The delta wing vortex bubble dataset with a grid size of 512^3, 1024^3, 2048^3, 4096^3, and 8192^3 takes 1.71, 2.25, 5.58, 20.65, and 113.16 seconds to construct the CPSO, respectively. The voxel count for the delta wing vortex bubble dataset is 114,215, 489,710, 2,594,110, 15,259,859, and 87,677,518 for grid sizes 512^3, 1024^3, 2048^3, 4096^3, and 8192^3, respectively. For an increase in dimensions from 512^3 to 1024^3, the number of voxels increases by 4.3x, and the amount of time to build the CPSO increases by 1.3x. For an increase in the grid size from 1024^3 to 2048^3, the number of voxels increases by 5.3x and the time to construct the CPSO increases by a factor of 2.5x. Raising the grid size from 2048^3 to 4096^3 increases the voxel count by 5.9x and the build time for the CPSO increases by 3.7x. Increasing the grid size from 4096^3 to 8192^3 increases the CPSO build time by 5.5x and the voxel count by 5.8x. For each increase in the grid size, both the CPSO and the number of voxels increases linearly at a similar rate. Like the ICE train and F6 aircraft datasets, it is a nonlinear increase in time for applying UFLIC. The increase in time for grid size 512^3 to 1024^3

is 4.8x, the increase in time for grid size 2048^3 is 5.7x, and the increase in time for grid size 4096^3 is 8.7x. For the increase in the gridsize to 8192^3, the amount of time to perform UFLIC increases by 19.7x. 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 aircraft: 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 6.1, which means that at least 99% of the dense grid is empty data, and the sparse octree removed those empty grid cells to save memory.

Finally, compared to the previous technique, the SFCPE [52], the CPSO scales beyond the SFCPE memory-limited 1024^3 grid size on the Nvidia Quadro K6000. The SFCPE 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 two-level grid from the SFCPE, 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 aircraft 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 SFCPE 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.

6.4 Summary

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 three dimensions to generate the two-dimensional 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. The sparse octree expands the ability of the closest point method to larger datasets, which is increasingly important as datasets 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.

CHAPTER 7

CONCLUSION

In this dissertation, we have presented several techniques for surface visualization. In Chapter 3, the red-black Gauss-Seidel advection scheme is introduced to place particles evenly on a surface. This pseudo-red-black Gauss-Seidel update scheme was applied to the GPU and achieved an order of magnitude speed-up over a similar CPU implementation.

However, this technique was not optimal for the GPU, and a new approach was sought to improve performance. A GPU Barnes-Hut tree code coupled with the closest point embedding is discussed in Chapter 4. The Barnes-Hut tree code was chosen to replace the binning structure and the Gauss-Seidel update because it is more efficient on the GPU. Further, the closest point embedding was used because the distance field required an iterative search for the surface, which was not a good mapping for the GPU. By coupling the Barnes-Hut tree code with the closest point embedding, the particles were mapped to hardware threads in a one-to-one manner.

The closet point embedding is a flexible data structure, and was adapted to surface flow visualization in Chapter 5. Previously, the surface was either parameterized or an image-space technique was used, with either being a compromise. The closest point embedding provides a flexible framework such that particle-based flow visualization techniques such as UFLIC were implemented without the difficulties of parameterization or the "popping" and self-occlusion problems of image-space based techniques.

Finally, the closest point embedding is a flexible data structure, but it scales poorly as the size increases. Therefore, the closest point sparse octree discussed in Chapter 6 is the first sparse octree structure introduced for the closest point method. This octree scales up to a $8,192^3$ grid size. Further, instead of a particle reprojection implementation of UFLIC, the *equivalence of gradience* is used to move the surface values into the sparse grid and allow for particles to advect, deposit values, and filter in three dimensions, which is, to our knowledge, the first implementation of an unsteady flow visualization technique using the closest point method.

In the future, we would like to explore the fast embedding technique to visualize unsteady flow on moving surfaces. Further, we would like to explore increasing the performance of the octree to bring the runtime down to near-interactive rates. In particular, adapting a two-level approach for the octree, such as [52], could improve the performance. Further, we would like to adapt other PDE-based flow visualization techniques, such as reaction-diffusion [95].

REFERENCES

[1] P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun, "Variational tetrahedral meshing," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 617–625. [Online]. Available: http://doi.acm.org/10.1145/1186822.1073238

[2] 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*, vol. 31, no. 6, pp. 1909–1923, 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03071.x

[3] J. Baert, A. Lagae, and P. Dutré, "Out-of-core construction of sparse voxel octrees," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13. New York, NY, USA: ACM, 2013, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2492045.2492048

[4] J. Barnes and P. Hut, "A hierarchical o(n log n) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, dec 1986.

[5] J. E. Barnes, "A modified tree code: Don't laugh; it runs," *Journal of Computational Physics*, vol. 87, no. 1, pp. 161 – 170, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002199919090232P

[6] H. Battke, D. Stalling, and H.-C. Hege, "Fast line integral convolution for arbitrary surfaces in 3d," in *Visualization and Mathematics*, H.-C. Hege and K. Polthier, Eds. Springer Berlin Heidelberg, 1997, pp. 181–195. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-59195-2_12

[7] J. Bédorf and S. Portegies Zwart, "A pilgrimage to gravity on gpus," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 201–216, 2012. [Online]. Available: http://dx.doi.org/10.1140/epjst/e2012-1647-6

[8] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational n-body code that runs entirely on the gpu processor," *J. Comput. Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[9] R. Belleman, J. Bdorf, and S. Portegies Zwart, "High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda," *New Astronomy*, vol. 13, no. 2, pp. 103–112, 2008, cited By (since 1996)108. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-35148867733&partnerID=40&md5=97b5cd93659bfa9fd310c14e3b9afeb5

[10] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989. [Online]. Available: http://dx.doi.org/10.1109/12.42122

[11] ——, "Prefix sums and their applications," Synthesis of Parallel Algorithms, Tech. Rep., 1990.

[12] D. Boltcheva, M. Yvinec, and J.-D. Boissonnat, "Mesh generation from 3d multi-material images," in *Medical Image Computing and Computer-Assisted Intervention MICCAI 2009*, ser. Lecture Notes in Computer Science, G.-Z. Yang, D. Hawkes, D. Rueckert, A. Noble, and C. Taylor, Eds. Springer Berlin Heidelberg, 2009, vol. 5762, pp. 283–290. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04271-3_35

[13] J. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[14] J. Bronson, J. Levine, and R. Whitaker, "Lattice cleaving: A multimaterial tetrahedral meshing algorithm with guarantees," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 20, no. 2, pp. 223–237, Feb. 2014.

[15] J. R. Bronson, J. A. Levine, and R. T. Whitaker, "Particle systems for adaptive, isotropic meshing of CAD models," in *19th IMR*, Oct. 2010, pp. 279–296.

[16] I. Buck *et al.*, "Brook for gpus: Stream computing on graphics hardware," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 777–786. [Online]. Available: http://doi.acm.org/10.1145/1186562.1015800

[17] 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*, ser. SIGGRAPH '93. New York, NY, USA: ACM, 1993, pp. 263–270. [Online]. Available: http://doi.acm.org/10.1145/166117.166151

[18] J. C. Caendish, D. A. Field, and W. H. Frey, "An apporach to automatic three-dimensional finite element mesh generation," *International Journal for Numerical Methods in Engineering*, vol. 21, no. 2, pp. 329–347, 1985. [Online]. Available: http://dx.doi.org/10.1002/nme.1620210210

[19] M. Callahan, M. Cole, J. Shepherd, J. Stinstra, and C. Johnson, "A meshing pipeline for biomedical models," *Engineering with Computers*, vol. 25, no. 1, pp. 115–130, 2009. [Online]. Available: http://www.sci.utah.edu/publications/callahan09/scirun_meshing.pdf

[20] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng, "Silver exudation," *J. ACM*, vol. 47, no. 5, pp. 883–904, Sep. 2000. [Online]. Available: http://doi.acm.org/10.1145/355483.355487

[21] 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*, ser. I3D '09. New York, NY, USA: ACM, 2009, pp. 15–22. [Online]. Available: http://doi.acm.org/10.1145/1507149.1507152

[22] P. Crossno and E. Angel, "Isosurface extraction using particle systems," in *IEEE Visualization 97*, 1997, pp. 495–498. [Online]. Available: http://doi.acm.org/10.1145/266989.267130

[23] N. Cuntz, A. Kolb, R. Strzodka, and D. Weiskopf, "Particle level set advection for the interactive visualization of unsteady 3d flow," *Comput. Graph. Forum*, vol. 27, no. 3, pp. 719–726, 2008.

[24] T. K. Dey and S. Goswami, "Tight cocone: a water-tight surface reconstructor," in *Proceedings of the eighth ACM symposium on solid modeling and applications*, ser. SM '03. New York, NY, USA: ACM, 2003, pp. 127–134. [Online]. Available: http://doi.acm.org/10.1145/781606.781627

[25] J. Dongarra *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342010391989

[26] Q. Du and M. Emelianenko, "Acceleration schemes for computing centroidal voronoi tessellations," *Numerical Linear Algebra with Applications*, vol. 13, no. 2-3, pp. 173–192, 2006. [Online]. Available: http://dx.doi.org/10.1002/nla.476

[27] Q. Du, V. Faber, and M. Gunzburger, "Centroidal voronoi tessellations: Applications and algorithms," *SIAM Rev.*, vol. 41, no. 4, pp. 637–676, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1137/S0036144599352836

[28] J. Dubinski, "A parallel tree code," *New Astronomy*, vol. 1, no. 2, pp. 133 – 147, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1384107696000097

[29] M. Edmunds *et al.*, "Surface-based flow visualization," *Computers & Graphics*, vol. 36, no. 8, pp. 974–990, 2012.

[30] E. Edwards and R. Bridson, "A high-order accurate particle-in-cell method," *International Journal for Numerical Methods in Engineering*, vol. 90, no. 9, pp. 1073–1088, 2012. [Online]. Available: http://dx.doi.org/10.1002/nme.3356

[31] D. Eppstein, "Global optimization of mesh quality." in *Tutorial at the 10th Int. Meshing Roundtable, Newport Beach.*, S. Shontz, Ed. Springer Berlin Heidelberg, 2001, pp. 367–384. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15414-0_22

[32] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of gpu algorithms for matrix-matrix multiplication," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, ser. HWWS '04. New York, NY, USA: ACM, 2004, pp. 133–137. [Online]. Available: http://doi.acm.org/10.1145/1058129.1058148

[33] D. F. Fisher, J. H. Delfrate, and D. M. Richwine, "In-flight flow visualization characteristics of the nasa f-18 high alpha research vehicle at high angles of attack," National Aeronautic and Space Agency (NASA), Tech. Rep., 1991. [Online]. Available: http://naca.larc.nasa.gov/search.jsp?R=19910010742&hterms=in-flight+visualization+characteristics&qs=N%3D0%26Ntk%3DAll%26Ntt%3Din-flight%2520visualization%2520characteristics%26Ntx%3Dmode%2520matchallpartial

[34] 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*, vol. 1, no. 2, pp. 133–141, Jun. 1995. [Online]. Available: http://dx.doi.org/10.1109/2945.468406

[35] L. A. Freitag and C. Ollivier-gooch, "Tetrahedral mesh improvement using swapping and smoothing," *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, vol. 40, no. 21, pp. 3979–4002, 1997.

[36] T. Fukushige *et al.*, "GRAPE-1A: Special-Purpose Computer for N-body Simulation with a Tree Code," *Publications of the Astronomical Society of Japan*, vol. 43, pp. 841–858, Dec. 1991.

[37] E. Gaburov, J. Bdorf, and S. P. Zwart, "Gravitational tree-code on graphics processing units: implementation in {CUDA}," *Procedia Computer Science*, vol. 1, no. 1, pp. 1119 – 1127, 2010, ¡ce:title¿ICCS 2010¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050910001250

[38] E. Gaburov, S. Harfst, and S. P. Zwart, "Sapporo: A way to turn your graphics cards into a grape-6," *New Astronomy*, vol. 14, no. 7, pp. 630 – 637, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1384107609000359

[39] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, "Interactive SPH simulation and rendering on the GPU," in *Proceedings ACM SIGGRAPH Eurographics Symposium on Computer Animation*, July 2010, pp. 55–64. [Online]. Available: http://www.zora.uzh.ch/43069/

[40] S. Green, "Particle simulation using cuda," NVIDIA, White Paper, May 2010.

[41] T. Hamada and K. Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.1

[42] P. S. Heckbert, "Fast surface particle repulsion," in *SIGGRAPH '97, New Frontiers in Modeling and Texturing Course*. ACM Press, 1997, pp. 95–114.

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

[44] Y. Hong, D. Zhu, X. Qiu, and Z. Wang, "Geometry-based control of fire simulation," *Vis. Comput.*, vol. 26, no. 9, pp. 1217–1228, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1007/s00371-009-0403-8

[45] J. Huang *et al.*, "Output-coherent image-space lic for surface flow visualization," in *Visualization Symposium (PacificVis), 2012 IEEE Pacific*, Feb 2012, pp. 137–144.

[46] B. Jobard, G. Erlebacher, and M. Hussaini, "Hardware-accelerated texture advection for unsteady flow visualization," in *Visualization 2000. Proceedings*, Oct 2000, pp. 155–162.

[47] B. Jobard, G. Erlebacher, and M. Y. Hussaini, "Lagrangian-eulerian advection for unsteady flow visualization," in *Proceedings of the Conference on Visualization '01*, ser. VIS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 53–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=601671.601678

[48] ——, "Lagrangian-eulerian advection of noise and dye textures for unsteady flow visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 8, no. 3, pp. 211–222, 2002.

[49] G. K. Karch, F. Sadlo, D. Weiskopf, C.-D. Munz, and T. Ertl, "Visualization of advection-diffusion in unsteady fluid flow," *Computer Graphics Forum*, vol. 31, no. 3pt2, pp. 1105–1114, 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03103.x

[50] M. Kim, G. Chen, and C. Hansen, "Dynamic particle system for mesh extraction on the gpu," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: ACM, 2012, pp. 38–46.

[51] M. Kim and C. Hansen, "GPU surface extraction with the closest point embedding," in *Proceedings of IS&T SPIE Visualization and Data Analysis, 2015*, February 2015. [Online]. Available: http://www.sci.utah.edu/publications/Kim2015b/Kim_SPIEVDA2015.pdf

[52] ——, "Surface flow visualization using the closest point embedding," *2015 IEEE Pacific Visualization Symposium*, April 2015. [Online]. Available: http://www.sci.utah.edu/publications/Kim2015a/Kim_PacVis2015.pdf

[53] ——, "Closest point sparse octree for surface flow visualization," 2016, in Submission.

[54] P. Kipfer, M. Segal, and R. Westermann, "Uberflow: a GPU-based particle engine," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA: ACM Press, 2004, pp. 115–122.

[55] A. Kolb and N. Cuntz, "Dynamic particle coupling for GPU-based fluid simulation," in *Proc. of the 18th Symposium on Simulation Technique*, 2005, pp. 722–727.

[56] A. Kolb, L. Latta, and C. Rezk-Salama, "Hardware-based simulation and collision detection for large particle systems," in *Proc. Graphics Hardware*, 2004, pp. 123–131.

[57] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3d flows," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, pp. 744–756, November 2005. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2005.87

[58] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *ACM SIGGRAPH 2003 Papers*, ser. SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 908–916. [Online]. Available: http://doi.acm.org/10.1145/1201775.882363

[59] S. Laine and T. Karras, "Efficient sparse voxel octrees," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 55–63. [Online]. Available: http://doi.acm.org/10.1145/1730804.1730814

[60] R. S. Laramee *et al.*, "The state of the art in flow visualization: Dense and texture-based techniques," *Computer Graphics Forum*, vol. 23, no. 2, pp. 203–221, 2004. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2004.00753.x

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

[62] E. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Supercomputing, ACM/IEEE 2001 Conference*, Nov 2001, pp. 43–43.

[63] 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*, ser. EUROVIS'06. Aire-la-Ville, Switzerland: Eurographics Association, 2006, pp. 29–34. [Online]. Available: http://dx.doi.org/10.2312/VisSym/EuroVis06/029-034

[64] ——, "Physically-based dye advection for flow visualization," *Computer Graphics Forum*, vol. 27, no. 3, pp. 727–734, 2008. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2008.01201.x

[65] 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*, vol. 14, no. 5, pp. 1067–1080, 2008.

[66] E. Lindholm, M. J. Kilgard, and H. Moreton, "A user-programmable vertex engine," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 149–158. [Online]. Available: http://doi.acm.org/10.1145/383259.383274

[67] Y. Liu *et al.*, "On centroidal voronoi tessellation&mdash;energy smoothness and fast computation," *ACM Trans. Graph.*, vol. 28, no. 4, pp. 101:1–101:17, Sep. 2009. [Online]. Available: http://doi.acm.org/10.1145/1559755.1559758

[68] Y. Liu, P. Foteinos, A. Chernikov, and N. Chrisochoides, "Multi-tissue mesh generation for brain images," in *Proceedings of the 19th International Meshing Roundtable*, S. Shontz, Ed. Springer Berlin Heidelberg, 2010, pp. 367–384. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15414-0_22

[69] Z. Liu and R. J. Moorhead, II, "Auflic: An accelerated algorithm for unsteady flow line integral convolution," in *Proceedings of the Symposium on Data Visualisation 2002*, ser. VISSYM '02. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 43–ff. [Online]. Available: http://dl.acm.org/citation.cfm?id=509740.509747

[70] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87. New York, NY, USA: ACM, 1987, pp. 163–169. [Online]. Available: http://doi.acm.org/10.1145/37401.37422

[71] C. B. Macdonald and S. J. Ruuth, "Level set equations on surfaces via the closest point method," *J. Sci. Comput.*, vol. 35, no. 2-3, pp. 219–240, 2008.

[72] ——, "The implicit closest point method for the numerical solution of partial differential equations on surfaces," *SIAM J. Scientific Computing*, vol. 31, no. 6, pp. 4330–4350, 2009.

[73] R. MacLeod *et al.*, "Subject-specific, multiscale simulation of electrophysiology: a software pipeline for image-based models and application examples," *Philosophical Transactions of The Royal Society A, Mathematical, Physical & Engineering Sciences*, vol. 367, no. 1896, pp. 2293–2310, 2009. [Online]. Available: http://www.sci.utah.edu/publications/Mac2009a/MacLeod_TRSA2009.pdf

[74] J. Makimo and M. Taiji, *Scientific Simulations with Special Purpose Computers: The Grade Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[75] J. Makino and P. Hut, "Galaxies in the connection machine," *Celestial mechanics*, vol. 45, no. 1-3, pp. 141–147, 1988. [Online]. Available: http://dx.doi.org/10.1007/BF01228995

[76] T. März and C. B. Macdonald, "Calculus on surfaces with general closest point functions," *SIAM J. Numerical Analysis*, vol. 50, no. 6, pp. 3303–3328, 2012.

[77] S. Mauch, "A fast algorithm for computing the closest point and distance transform," Calif. Inst. of Technology, Pasadena, CA, USA, Tech. Rep., 2000.

[78] N. Max, B. Becker, and R. Crawfis, "Flow volumes for interactive vector field visualization," in *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, Oct 1993, pp. 19–24.

[79] M. Meyer, P. Georgel, and R. Whitaker, "Robust particle systems for curvature dependent sampling of implicit surfaces," in *Proceedings of the International Conference on Shape Modeling and Applications (SMI)*, June 2005, pp. 124–133. [Online]. Available: http://www.sci.utah.edu/publications/miriah05/smi05meyer.pdf

[80] M. Meyer, R. Kirby, and R. Whitaker, "Topology, accuracy, and quality of isosurface meshes using dynamic particles," *IEEE Transactions on Visualization and Computer Graphics (Visualization 2007)*, vol. 13, no. 6, pp. 1704–1711, 2007. [Online]. Available: http://www.sci.utah.edu/publications/meyer07/vis07meyer.pdf

[81] M. Meyer, R. Whitaker, R. Kirby, C. Ledergerber, and H. Pfister, "Particle-based sampling and meshing of surfaces in multimaterial volumes," *IEEE Transactions on Visualization and Computer Graphics (Visualization 2008)*, vol. 14, no. 6, pp. 1539–1546, 2008. [Online]. Available: http://www.sci.utah.edu/publications/meyer08/Meyer_VCG2008.pdf

[82] T. Moller, K. Mueller, Y. Kurzion, R. Machiraju, and R. Yagel, "Design of accurate and smooth filters for function and derivative reconstruction," in *Volume Visualization, 1998. IEEE Symposium on*, Oct 1998, pp. 143–151.

[83] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[84] K. Nitadori, J. Makino, and P. Hut, "Performance tuning of n-body codes on modern microprocessors: I. direct integration with a hermite scheme on x86_64 architecture," *New Astronomy*, vol. 12, no. 3, pp. 169 – 181, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1384107606000947

[85] Nvidia, "Tesla k80 is all about instant gratification, early users say," 2014. [Online]. Available: http://blogs.nvidia.com/blog/2014/11/18/tesla-k80-perf/

[86] NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[87] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x

[88] V. N. Parthasarathy, C. M. Graichen, and A. F. Hathaway, "A comparison of tetrahedron quality measures," *Finite Elem. Anal. Des.*, vol. 15, no. 3, pp. 255–261, Jan. 1994. [Online]. Available: http://dx.doi.org/10.1016/0168-874X(94)90033-7

[89] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko, "Function representation in geometric modeling: Concepts, implementation and applications," *The Visual Computer*, vol. 11, no. 8, pp. 429–446, 1995.

[90] J.-P. Pons *et al.*, "High-quality consistent meshing of multi-label datasets," in *Information Processing in Medical Imaging*, ser. Lecture Notes in Computer Science, N. Karssemeijer and B. Lelieveldt, Eds. Springer Berlin Heidelberg, 2007, vol. 4584, pp. 198–210. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73273-0_17

[91] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '02. New York, NY, USA: ACM, 2002, pp. 703–712. [Online]. Available: http://doi.acm.org/10.1145/566570.566640

[92] J. Revelles, C. Urea, and M. Lastra, "An efficient parametric algorithm for octree traversal," in *Journal of WSCG*, 2000, pp. 212–219.

[93] S. J. Ruuth and B. Merriman, "A simple embedding method for solving partial differential equations on surfaces," *J. Comput. Physics*, vol. 227, no. 3, pp. 1943–1961, 2008.

[94] P. Salz, A. Reske, H. Wrigge, G. Scheuermann, and H. Hagen, "Improving Electrical Impedance Tomography Imaging of the Lung with Patient-specific 3D Models," in *Visualization in Medicine and Life Sciences*, L. Linsen, H. C. Hege, and B. Hamann, Eds. The Eurographics Association, 2013.

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

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

[97] M. Schwarz and H.-P. Seidel, "Fast parallel surface and solid voxelization on gpus," in *ACM SIGGRAPH Asia 2010 Papers*, ser. SIGGRAPH ASIA '10. New York, NY, USA: ACM, 2010, pp. 179:1–179:10. [Online]. Available: http://doi.acm.org/10.1145/1866158.1866201

[98] SCI, bioMesh3D: Quality Mesh Generator for Biomedical Applications. Scientific Computing and Imaging Institute (SCI). [Online]. Available: http://www.biomesh3d.org

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

[100] H.-W. Shen, C. R. Johnson, and K.-L. Ma, "Visualizing vector fields using line integral convolution and dye advection," in *Proceedings of the 1996 Symposium on Volume Visualization*, ser. VVS '96. Piscataway, NJ, USA: IEEE Press, 1996, pp. 63–ff. [Online]. Available: http://dl.acm.org/citation.cfm?id=236226.236234

[101] H.-W. Shen and D. L. Kao, "A new line integral convolution algorithm for visualizing time-varying flow fields." *IEEE Trans. Vis. Comput. Graph.*, vol. 4, no. 2, pp. 98–108, 1998. [Online]. Available: http://dblp.uni-trier.de/db/journals/tvcg/tvcg4.html#ShenK98

[102] J. R. Shewchuk, "Unstructured mesh generation," in *In Combinatorial Scientific Computing*, U. Naumann and O. Schenk, Eds. CRC Press, 2012, pp. 257–297.

[103] ——, "Constrained delaunay tetrahedralizations and provably good boundary recovery," in *In Eleventh International Meshing Roundtable*, 2002, pp. 193–204.

[104] K. Shimada and D. C. Gossard, "Bubble mesh: Automated triangular meshing of non-manifold geometry by sphere packing," in *Proceedings of the Third ACM Symposium on Solid Modeling and Applications*, ser. SMA '95. New York, NY, USA: ACM, 1995, pp. 409–419. [Online]. Available: http://doi.acm.org/10.1145/218013.218095

[105] D. Stalling and H.-C. Hege, "Fast and resolution independent line integral convolution," in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '95. New York, NY, USA: ACM, 1995, pp. 249–256. [Online]. Available: http://doi.acm.org/10.1145/218380.218448

[106] 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*, ser. VISSYM'04. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2004, pp. 221–230. [Online]. Available: http://dx.doi.org/10.2312/VisSym/VisSym04/221-230

[107] D. Swenson, private communication, 2012.

[108] D. Swenson, J. Levine, Z. Fu, J. Tate, and R. MacLeod, "The effect of non-conformal finite element boundaries on electrical monodomain and bidomain simulations," *Computing in Cardiology*, no. 37, pp. 97–100, 2010. [Online]. Available: http://www.sci.utah.edu/publications/swenson10/Swenson_CinC2010.pdf

[109] L. L. Tian, C. B. Macdonald, and S. J. Ruuth, "Segmentation on surfaces with the closest point method," in *ICIP*, 2009, pp. 3009–3012.

[110] J. J. van Wijk, "Image based flow visualization," in *SIGGRAPH*, 2002, pp. 745–754.

[111] ——, "Image based flow visualization for curved surfaces," in *IEEE Visualization*, 2003, pp. 123–130.

[112] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 12–21. [Online]. Available: http://doi.acm.org/10.1145/169627.169640

[113] 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*, 2003, pp. 107–114.

[114] D. Weiskopf, "Dye advection without the blur: A level-set approach for texture-based visualization of unsteady flow," *Comput. Graph. Forum*, vol. 23, no. 3, pp. 479–488, 2004.

[115] R. T. Whitaker, "Reducing aliasing artifacts in iso-surfaces of binary volumes," in *Proceedings of the 2000 IEEE symposium on volume visualization*, ser. VVS '00. New York, NY, USA: ACM, 2000, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/353888.353893

[116] A. Witkin and P. Heckbert, "Using particles to sample and control implicit surfaces," *Computer Graphics*, pp. 269–278, Jul. 1994, proceedings of SIGGRAPH'94.

[117] R. Yokota and L. Barba, "Hierarchical n-body simulations with autotuning for heterogeneous systems," *Computing in Science Engineering*, vol. 14, no. 3, pp. 30–39, 2012.

[118] Y. Zhang and C. Bajaj, "3d finite element meshing from imaging data," *Computer Methods in Applied Mechanics and Engineering (CMAME)*, pp. 5083 – 5106, 2005.

[119] Y. Zhang, T. J. Hughes, and C. L. Bajaj, "An automatic 3d mesh generation method for domains with multiple materials," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 58, pp. 405 – 415, 2010, computational Geometry and Analysis. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S004578250900214X

[120] M. Zöckler, D. Stalling, and H.-C. Hege, "Parallel line integral convolution," *Parallel Computing*, vol. 23, no. 7, pp. 975 – 989, 1997, parallel graphics and visualisation. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819197000392