ABSTRACT

Title of dissertation: HIGHLY PARALLEL GEOMETRIC CHARACTERIZATION AND VISUALIZATION OF VOLUMETRIC DATA SETS

Derek Christopher Juba

Dissertation directed by: Professor Amitabh Varshney
Department of Computer Science

Volumetric 3D data sets are being generated in many different application areas. Some examples are CAT scans and MRI data, 3D models of protein molecules represented by implicit surfaces, multi-dimensional numeric simulations of plasma turbulence, and stacks of confocal microscopy images of cells. The size of these data sets has been increasing, requiring the speed of analysis and visualization techniques to also increase to keep up.

Recent advances in processor technology have stopped increasing clock speed and instead begun increasing parallelism, resulting in multi-core CPUS and many-core GPUs. To take advantage of these new parallel architectures, algorithms must be explicitly written to exploit parallelism. In this thesis we describe several algorithms and techniques for volumetric data set analysis and visualization that are amenable to these modern parallel architectures.

We first discuss modeling volumetric data with Gaussian Radial Basis Functions (RBFs). RBF representation of a data set has several advantages, including lossy compression, analytic differentiability, and analytic application of Gaussian blur. We also describe a parallel volume rendering algorithm that can create images of the data directly from the RBF representation.

Next we discuss a parallel, stochastic algorithm for measuring the surface area of volumetric representations of molecules. The algorithm is suitable for implementation on a GPU and is also progressive, allowing it to return a rough answer almost immediately and refine the answer over time to the desired level of accuracy.

After this we discuss the concept of Confluent Visualization, which allows the visualization of the interaction between a pair of volumetric data sets. The interaction is visualized through volume rendering, which is well suited to implementation on parallel architectures.

Finally we discuss a parallel, stochastic algorithm for classifying stem cells as having been grown on a surface that induces differentiation or on a surface that does not induce differentiation. The algorithm takes as input 3D volumetric models of the cells generated from confocal microscopy. This algorithm builds on our algorithm for surface area measurement and, like that algorithm, this algorithm is also suitable for implementation on a GPU and is progressive.

HIGHLY PARALLEL GEOMETRIC CHARACTERIZATION AND VISUALIZATION OF VOLUMETRIC DATA SETS

by

Derek Christopher Juba

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Foreword

This document contains work which was co-authored with other colleagues. I certify that I have made substantial contributions to all jointly authored work included in this document.

Acknowledgments

I would like to thank my advisor, Professor Amitabh Varshney, for his support during my time in the graduate program and for sticking with me even when the going was rough.

I would also like to thank my family for their continuous support and encouragement throughout my academic career.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern technology has given rise to the generation of 3D volumetric data sets in many application areas. CAT scans and MRI data are being generated by the medical field. 3D models of protein molecules represented by implicit surfaces are being generated by computational biologists. Physicists are generating multi-dimensional numeric simulations of plasma turbulence. Confocal microscopy technology is allowing biologists to capture stacks of cell images at varying depths, which can then be combined to create a 3D model of the cell.

Because these data sets can be large and can have a need for high-throughput processing, we would like to leverage the most recent advances in high-performance computing for their analysis. Recent advances in processor technology have stopped increasing clock speed and instead begun increasing parallelism, resulting in multi-core CPUS and many-core GPUs.

To take advantage of these new parallel architectures, algorithms must be explicitly written to exploit parallelism. We have found that progressive stochastic algorithms map well to these architectures. With these algorithms each parallel processor can compute one stochastic sample– scaling to more processors then simply results in an increase in the rate at which samples are taken. An additional benefit is that the progressive nature of these algorithms allows an initial estimated result to be produced quickly, and then refined over time. The threads also tend to be independent, requiring little or no inter-thread communication.

We have applied these techniques and algorithms to several different areas. The first is fitting scalar fields with Gaussian radial basis functions in the context of medical imaging and physical simulations. In addition, we have done work on using parallel, stochastic algorithms to measure molecular surface area. We have also done work in confluent visualization– using one data set to control the visualization of another– in the context of plasma physics. Finally, we have done work in the area of stem cell therapy, using these algorithms to identify whether or not a cell has differentiated into a desired type of tissue.

1.1 Gaussian RBF Fitting

3D volumetric data sets are often represented as simple scalar fields. This is sufficient for many purposes, but some applications can benefit from representing the data as a mixture of Gaussian Radial Basis Functions (RBFs). Gaussian RBFs are easily differentiable, which allows certain derived quantities such as the curvature of an implicit surface to be computed analytically. A Gaussian RBF representation can also have a Gaussian blur applied analytically by simply adjusting the parameters of each RBF.

To fit RBFs to data, we make use of our work on Gaussian RBF modelling and rendering [2]. In this work we fit the RBFs using a multi-resolution octree hierarchy. We first fit a single RBF to the highest level of the octree. We then evaluate the fitting error, and if it is too high, we subdivide the octree node into eight children and fit an RBF to each child. We then evaluate the fitting error of each of the children and continue the subdivision and fitting until the error drops below a given threshold.

To fit an RBF to an octree node we must determine the RBF weight, width, and center. The center of the RBF is set to the location of maximum error. The weight is then set such that the error at the center is zero. The anisotropic width can be computed using several methods. One method involves incrementally increasing the width along each axis until the increases stop decreasing the fitting error. Another method involves setting the width non-iteratively using Maximum Likelihood Estimation.

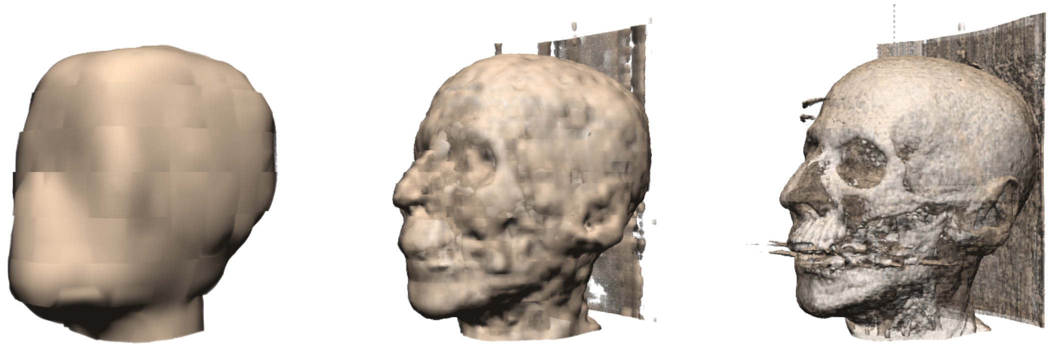


Figure 1.1: Renderings from three different levels of an RBF hierarchy for the UNC Head data set. From left to right, the renderings were generated using 4, 6, and 8 levels from the octree and consist of 561, 20.6 K, and 485 K RBFs.

In addition to the fitting algorithm, we also describe a method of rendering a fitted data set directly from the RBF representation. We perform direct volume rendering by incrementally stepping along rays shot from an eye position through the data set. At each step along the ray we evaluate the RBFs in the hierarchy at that point and use this value to get a color and opacity from a transfer function. We use the colors and opacities along each ray to produce the final color of a pixel in the image.

The octree hierarchy allows us to make several optimizations in the rendering. One optimization we can make is if the sample point along the ray is far from the eye position, we can produce a lower resolution version of the data by not including RBFs from all levels of the octree, instead stopping the evaluation of the RBFs at some level above the leaves. Another optimization is that we can skip over octree cells that do not contain a range of values that would give color or opacity with the current transfer function.

1.2 Parallel Stochastic Measurement

Some applications require measuring geometric properties of a 3D object such as its surface area or volume. For applications where the data is large or high-throughput is required, it would be beneficial to perform these measurements using a parallel algorithm. Such an algorithm is described in our work on measuring the surface area of molecules by intersecting the surface of the molecule with a set of random lines [3].

In this work we model molecules as a sum of Gaussian radial basis functions, with each atom being represented with a single RBF. The representation of atoms by Gaussian RBFs is inspired by the electron cloud around each atom. Unlike our previously described work on Gaussian RBF fitting we do not do the fitting hierarchically, instead using a simple bucketing spacial data structure. RBF centers are set to atom centers, and RBF weights and widths are set based on atomic radius so that an isosurface of the RBF corresponds to a van der walls surface of the atom.

Once we have the RBF model of the molecule, we create a set of random lines to intersect it with by choosing pairs of points on the surface of a bounding sphere. We uniformly generate points on the surface of the sphere by using a quasi-random sequence of numbers to give the point coordinates. Each pair of points then determines a line.

To count the number of intersections of each line with the molecule we incrementally step along each line evaluating the RBFs representing the molecule's atoms at that point. We determine whether each point is inside or outside the molecule by comparing the value of the RBFs with an isovalue. If a point is found to be inside the molecule when the previous point was outside, or vice versa, an intersection with the molecular surface is recorded.

Once we have the count of intersections with the molecular surface, we can use the Cauchy-Crofton formula from integral geometry to estimate the molecular surface area. This formula relates the surface area and number of line intersections with a bounding sphere to the surface area and number of intersections with the enclosed molecular surface. Since the surface area of the bounding sphere, the number of intersections with the bounding sphere, and the number of intersections with the molecular surface area are known, the area of the molecular surface can be estimated.

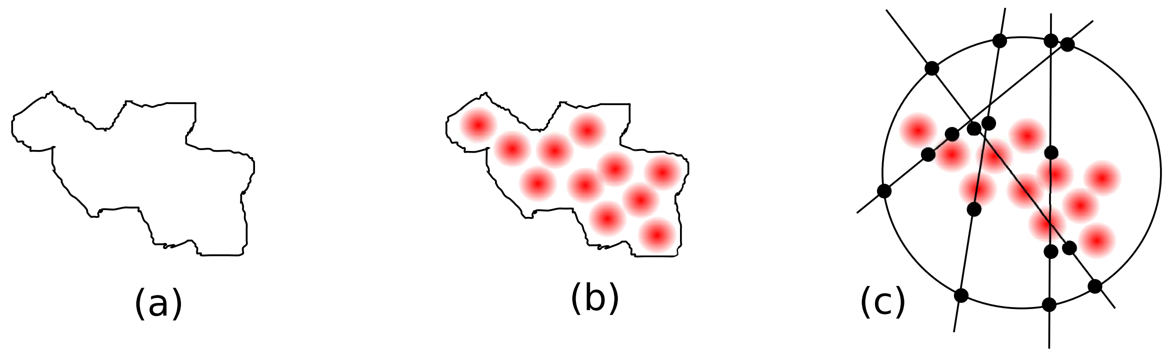


Figure 1.2: Given a scalar field data set such as a cell or molecule (part a) we fit the data with Gaussian radial basis functions (part b) then intersect the data and a bounding sphere with a set of random lines, counting the number of intersections (part c).

This algorithm has several nice features. First, it is easily parallelizable since each line's intersections with the molecule can be calculated independently. Our parallel GPU implementation is 3x – 10x faster than existing CPU implementations for the same error level. Second, it is incremental, meaning that it returns a rough estimate of the surface almost immediately, and gives gradually more accurate estimates as more lines are intersected. This is useful since in some applications the rough estimate might be used to determine whether or not additional time should be spent to compute the more refined estimate.

1.3 Confluent Visualization

Some visualization applications involve understanding the spatial relationship between a pair of superimposed data sets. One example is the phenomenon of stem cell differentiation being induced by surface geometry, which involves a data set of the cell itself and a data set of the surface on which it is resting. Interesting information may be derived by looking at the locations where the two data sets interact.

Another example is physical simulations of gyrokinetic turbulence performed in the context of research on fusion energy. These simulations can produce complex, high dimensional data sets that are difficult to visualize. In order to make the data more amenable to visualization, derived quantities are often visualized instead– in this case heat flux and electric potential.

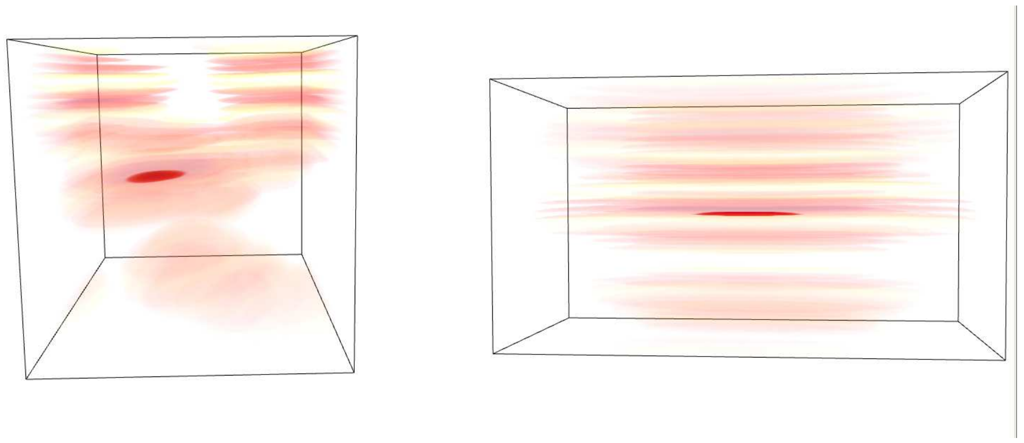


Figure 1.3: Example of Confluent Visualization in gyrokinetic turbulence simulation. Visible structures indicate zones of high heat flux. Color represents values of the electric potential.

To visualize these quantities we use standard direct volume rendering with a color and opacity transfer function. However, unlike in traditional direct volume rendering where the same value from the scalar field would be used to generate both the color and opacity, we use electric potential to generate the color and heat flux to generate the opacity. The result is that values of electric potential are visualized in areas of high heat flux. We refer to this as Confluent Visualization [4].

1.4 Regenerative Medicine

Tissue transplants are used to treat disorders of many different organs and tissues, including the heart, the bladder, and the urethra. Unfortunately there are several drawbacks of current transplant technology, such as requiring patients to be on immunosuppresant drugs for the remainder of their life. Patients must also compete for a limited supply of donor organs. These and other drawbacks of transplants have created a large market for stem cell and regenerative medicine.

Stem cells are cells which have the ability to differentiate and grow into multiple different types of organ and tissue. One issue with stem cells is how to get them to differentiate into the desired cell type. This can be accomplished through exposure to various chemicals, but one current line of work has shown that in some cases stem cells can be made to differentiate simply by placing them on a surface of a certain texture.

When working with stem cells it is useful to be able to tell whether a given cell population has differentiated or not. We have developed an algorithm based on our work on molecular surface area measurement [3] that can identify whether a cell was grown on a differentiating or non-differentiating surface with over 80% accuracy. The algorithm works on a 3D model of the cell which can be generated using confocal microscopy.

As with our molecular measurement work, this algorithm begins by intersecting

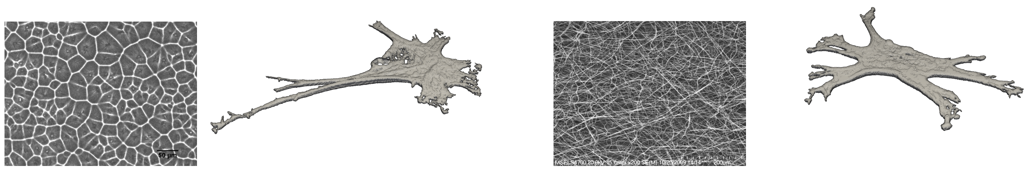


Figure 1.4: Scanning Electron Micrographs (SEM) of surfaces upon which stem cells can be grown and confocal microscopy models of cells grown on the surfaces. Left to right: SEM of spun-coat film surface that does not induce differentiation. Cell grown on spun-coat film. SEM of nanofiber scaffold surface that does induce differentiation. Cell grown on nanofiber scaffold.

the 3D model of the cell with a set of random lines. The lengths of the line segments that pass through the interior of the cell are stored in a histogram. These histograms can be used as data points to train a machine learning algorithm, such as a Support Vector Machine. Once the algorithm is trained, histograms of query cells can be classified as coming from either a differentiating or non-differentiating surface.

Like our molecular measurement algorithm, this algorithm is parallel and progressive, making it suitable for future high-throughput cell classification applications. Classification of the cell by analysis of its shape also allows the cell to be classified at an earlier stage of development than is possible by other means such as chemical staining.

Chapter 2

Gaussian RBF Fitting

2.1 Introduction

In this chapter we discuss modelling volumetric data with Gaussian Radial Basis Functions (RBFs). Representation of volumetric data with Gaussian RBFs has several advantages. First, the RBF representation may be smaller than the original data, resulting in lossy compression. This reduction in data size can reduce memory accesses resulting in a performance increase. Second, Gaussian RBFs are easily differentiable, which allows certain derived quantities such as the curvature of an implicit surface to be computed analytically. Third, a Gaussian RBF representation can also have a Gaussian blur applied analytically by simply adjusting the parameters of each RBF.

In addition to the RBF representation, we describe a parallel, GPU-accelerated volume rendering algorithm that can generate images of the data directly from the RBF representation.

The following material in this chapter applies the concepts of RBF fitting to medical and physics data, but the concepts are general enough that they could be applied to any type of volumetric data. The material in this chapter has been previously published [2].

Scientific visualization is currently facing a grand challenge in coping with vast quantities of data arising from high-fidelity acquisitions and large-scale scientific simulations. Such datasets range from a few gigabytes to several terabytes and are often characterized by an imperative need for interactive exploratory visualization capabilities. For instance, the Richtmyer-Meshkov instability simulation performed at the Lawrence Livermore National Laboratory [5] has produced a $2048\times2048\times1920$ data over 273 time steps. The sheer size of this data makes it a challenge to visualize it on commodity graphics hardware.

We believe that implicit representations offer a powerful model for facilitating interactive visual exploration of large volumetric datasets. Implicit functions have been used for almost two decades in visual computing. Introduced as *blobby models* [6] and *metaballs* [7], they have grown to be widely used in games and movies. A nice overview of some of the early work in implicit surfaces can be found in [8]. The use of implicit representations for volume modeling and rendering is a recent phenomenon. Implicit representations offer several advantages for volumes. First, their expressive power makes them well-suited for trading off memory accesses with computation. This is proving to be a powerful technique to hide memory latency for modern multi-core architectures. Second, the analytical formulation of implicit representations is highly amenable to local geometry processing, such as computing first and higher-order derivatives. Indeed, recent research has established the relationship of such geometric operators to features such as vortices and shocks by Weiler *et al.* [9] and the principled design of transfer functions by Kniss *et al.* [10, 11]. Third, volumetric implicit representations offer a multi-scale, and view-dependent capability to control visual detail in an intuitive manner. This is likely to have a direct implication in

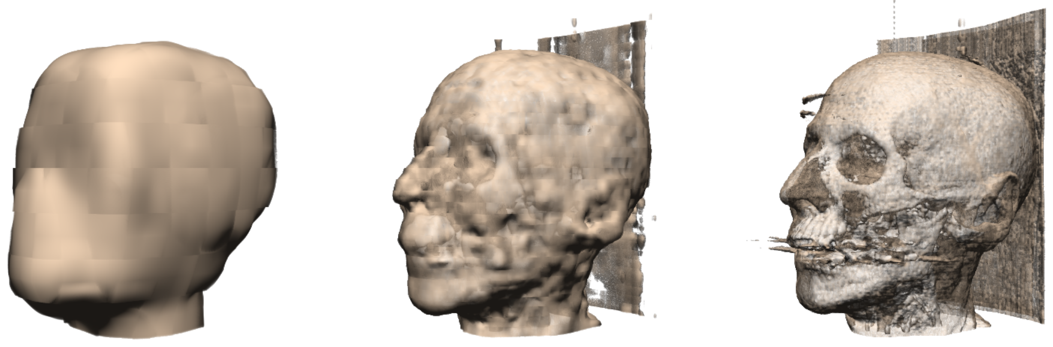


Figure 2.1: Renderings from three different levels of an RBF hierarchy for the UNC Head data set. From left to right, the renderings were generated using 4, 6, and 8 levels from the octree and consist of 561, 20.6 K, and 485 K RBFs.

devising techniques for facilitating comprehension and reducing clutter in visual depictions.

In this chapter we present an algorithm for succinctly representing and efficiently rendering large scalar volumetric data using a hierarchy of implicit functions based on anisotropic radial basis functions. The novel contributions of our work are:

1. A multiresolution representation using anisotropic radial basis functions that can encode a given volumetric dataset with progressively greater detail. Our representation is well-suited for time-critical rendering, progressive refinement, view-dependent level-of-detail, and progressive transmission.

2. An $O(n \log n)$-scalable fitting algorithm based on Maximum Likelihood Estimation that can rapidly fit large datasets in a memory-friendly manner. Specifically, we can fit 512^3 dataset in around 20 minutes with a 1.6% RMS error.

3. A GPU-based ray-casting algorithm that can efficiently and directly render from the hierarchical implicit representation of volumes. Our GPU-based ray-casting algorithm supports acceleration techniques such as empty-space skipping and early-ray termination with implicit volumes.

4. A multiresolution octree hierarchy over implicit RBFs that can leverage prior work on octree location codes for efficient ray-casting, obviates the need to store cell boundaries, and enables view-dependent level-of-detail rendering.

We review related work in section 2.2. We present our fitting algorithm in section 2.3 and our direct volume rendering algorithm in section 2.4. Finally, we conclude with some suggestions for future work in section 2.5.

2.2 Related Work

The basic goal of implicit function fitting is to fit a representation $f(x_i) = d_i, \quad i = 1, \ldots, n$, to surface or volumetric data, where d_i are appropriately chosen scalars, such that the level sets of the function f, given by the values d_i have some meaning associated either with the geometry or the properties of the object. When fitting an implicit function to the input data, a local form of the fitting expression is desirable, since the fit must adapt to local geometrical features. Radial basis functions (RBFs) have been shown to be a versatile fitting tool in various fields. A strong mathematical basis for their theory has been established and theorems showing accuracy in various normed spaces have been proven for both surface and volumetric data [12, 13, 14, 15, 16, 17, 18, 19]. These representations have several nice properties associated with sensitivity to local features, ability to support different levels of detail, ability to mitigate noise, ability to incorporate various degrees of smoothness and other priors via regularization and choice of particular RBFs (e.g., thin plate splines [20]). While these methods are considered to be accurate, their naive implementations are both expensive to fit, and subsequently evaluate, making them not very popular methods for fitting large datasets of the type we are dealing with in this chapter.

Research by Beatson *et al.* [21] shows how RBF function fitting and evaluation of the fitted function can be sped up by an order of magnitude using the fast multipole method (FMM) [22]. They used non-compactly supported RBFs due to their interpolation and extrapolation properties. Morse *et al.* [15] were the first to fit surfaces using compactly-supported RBFs. Due to their local region of influence, compactly-supported RBFs can permit efficient evaluation and the ability to incorporate local changes to the fitted function without the need for the FMM data-structures. Co *et al.* [23] and Jang *et al.* [24] were the first to represent scattered and irregular volumetric scalar fields using RBFs. They used Principal Component Analysis (PCA) [25] to cluster and determine centers for the Gaussian RBFs and used the Levenberg-Marquardt optimization method to determine the Gaussian RBF variances. Weiler *et al.* [9] presented a k-d-tree-based method to fit Gaussian RBFs to an unstructured volumetric vector field. In addition to using PCA clustering, they select some of their RBF centers to be at peaks and troughs of low frequencies in the data. They also use an approximate iterative method to quickly solve the system of equations for the RBF weights. Hong *et al.* [26] use arbitrarily-oriented elliptical RBFs to fit data on an irregular grid. The RBF variances and orientations are chosen to match the Voronoi cells of the data points. PCA is used to create an initial guess for the RBF parameters, which is then refined using an iterative optimization algorithm. Jang *et al.* [27] give a method of fitting arbitrarily-oriented elliptical RBFs using non-linear optimization, and extend it to support vector data.

Rendering of implicit functions has a rich history in visual computing. Ray tracing of implicit surfaces has been reasonably well-studied [6, 28, 29, 30, 31]. An alternative to ray-tracing is to sample the implicit surface with a collection of well-distributed particles and then render such particles. Methods for carefully sampling the implicit surfaces have been discussed by Witkin and Heckbert [32] as well as by Turk and O'Brien [18]. Another approach involves converting the implicit functions to polygons by using marching cubes [33] or continuation methods [34, 35]. The polygons can then be rendered as in traditional graphics. Direct volume rendering of implicit functions is a relatively new endeavor. Jang *et al.* [24] and Weiler *et al.* [9] have developed GPU-based volume rendering algorithms that proceed in a slice-by-slice fashion. For each fragment in a slice, a fragment program iterates over RBF parameters stored in a texture, computes the scalar value at that location, and looks up the corresponding color from a 1D texture. Neophytou *et al.* [36] present a splatting-based GPU-accelerated volume rendering method for arbitrarily-oriented elliptical RBFs. Their method splats each RBF onto each intersecting slice as a textured polygon and accumulates the splats in a texture buffer. The slice can then be rendered after classification with a fragment program.

In this chapter we present the first GPU-accelerated ray-casting algorithm for direct rendering of implicit volumes and the first octree-structured RBF representation of regular volume data that lends itself well to adaptive and progressive levels of detail.

2.3 Modelling

We model volume data as a sum of Radial Basis Functions (RBFs). Symbolically, this can be represented as:

$$f(x) = w_0 + \sum_{i=1}^{M} w_i \, \phi_i(x) \tag{2.1}$$

where

x	is the position vector of a point in the volume
$f(x)$	is the scalar value at that point
ϕ_i	is the ith RBF
M	is the number of RBFs
w_0	is a constant term
w_i	is the weight of the ith RBF

In this work we have used Gaussian RBFs:

$$\phi_i(x) = e^{-r^2/2} \tag{2.2}$$

2.3.1 Multiresolution Hierarchical Fitting

Since we are targeting regular volumetric data, we can take advantage of the inherent structure of the data and fit the RBFs to a multi-resolution octree hierarchy, from low resolution to high. We start by fitting a single RBF to a 2^3 resolution downsampled version of the given dataset. This becomes the root of our RBF hierarchy. We then take a version of the dataset that has been downsampled to 4^3, evaluate the initial (root) RBF at each of the 4^3 data points, and compute the difference between the root RBF value and the downsampled version. The 4^3 block of residual data is then divided into eight blocks of 2^3 data values, and the entire process is repeated for each of these blocks. This is continued using versions of the data that have been downsampled to 8^3, 16^3, etc. (see fig. 2.2). Each time a new block is fit, the RBFs in that block and all of its ancestors are sampled at the locations of the data points in the full resolution data, and the current fitting error is computed. If the error is below a user-defined threshold, that block is not subdivided any further. Otherwise, the blocks will continue to be subdivided until they reach the full original data resolution.

This fitting algorithm only ever needs to access the full data in a sequential manner, during the error evaluation stage. All other operations are performed on one small, fixed-size block of data at a time, which will likely fit in a memory cache. Besides being memory-friendly, the overall fitting algorithm also scales in a $\mathrm{O}(n \log n)$ manner.

In this method we truncate each RBF at the boundary of its octree cell. Due to fitting errors, this would cause noticeable discontinuities at the borders between octree cells during rendering. To avoid this, we fit each RBF to not only the data within its 2^3 block, but also to data within a 1 data point border around that block (each RBF is thus fitted to 4^3 data values). The fitting area of each block therefore

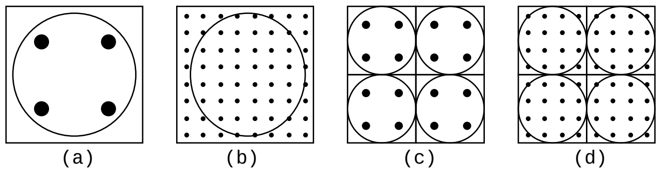


Figure 2.2: Overview of the fitting process. An RBF is first fit to low-resolution data (a), and the fitting error is evaluated at the full resolution (b). If the error is too high, additional RBFs are fit at a higher resolution (c), and their error evaluated (d). In the actual implementation the borders of each block would overlap neighboring blocks by one data point on each side, giving each block a resolution of 4^3 rather than 2^3 – this is omitted in this illustration for clarity.

overlaps half of each face-adjacent block, one quarter of each edge-adjacent block, and one eighth of each corner-adjacent block (see fig. 2.3 for an example on a quadtree). During rendering, the RBF values are blended together based on the distance from the center of each RBF's block (see section 2.4.3).

2.3.2 Basic Algorithm

In the simplest version of the algorithm the RBF widths σ_i are identical. In this case, the RBF radius r is simply:

$$r = \|\frac{x - \mu_i}{\sigma_i}\| \tag{2.3}$$

where x is the position vector of a point in the volume, μ_i is the position vector of the ith RBF's center, and σ_i defines the width of the i^{th} RBF.

Defining the RBF approximation is then only a matter of choosing a constant term w_0, RBF weights w_i, RBF centers μ_i, and RBF widths σ_i. Some results of

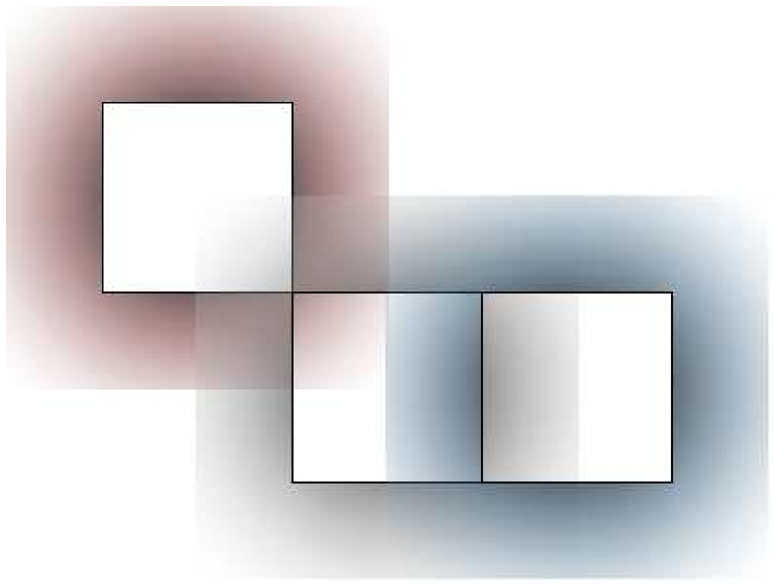


Figure 2.3: Blending between quadtree nodes. Only one edge-adjacent node (blue) and one corner-adjacent node (red) are shown. Sample points within the central block (grey) would be blended with samples from adjacent blocks whose shaded regions the points fell in.

this simple algorithm are shown in the first row of table 2.2. Some possible choices for the initial value of the constant term are zero, the minimum data value, or the mean of the data. In our experiments we found that setting the initial value to zero often produced the best results – this was done for all results reported in this work.

To compute the RBF centers and weights, the following simple algorithm can be used:

1. Perform a linear search over the data to find the data point with the maximum approximation error. This will be the location of the new RBF's center.

2. Set the RBF's weight to the error value at this cell. Since Gaussian RBFs have a value of 1 at their center, setting the weight in this way will cause the

center data point to have zero error.

3. Update error values stored at each data point within the RBF's radius of influence.

4. Repeat this process until the desired error threshold is reached.

2.3.3 Anisotropic Width Selection

A natural extension to the basic algorithm is to allow the widths of the RBFs to vary anisotropically. This can be accomplished by making σ in equation 2.1 a vector and allowing it to take on different values for different dimensions, dividing it componentwise into the $x - \mu_i$ vector. General elliptical RBFs were used by Hong *et al.*[26], Jang *et al.*[27], and Neophytou *et al.*[36], although we restrict our RBFs to be axis-aligned, sacrificing some generality for a more compact representation. The effect of using anisotropic widths is illustrated in table 2.1.

The simplest way to select the RBF widths is by using a greedy algorithm. For each RBF, we first initialize the value of each $\sigma_{xi}, \sigma_{yi}, \sigma_{zi}$ to some small number (we use 0.3 times the width of a grid cell). We then begin iterating repeatedly over the x, y, z dimensions. For each dimension, we increment the corresponding σ_i by some small number (we use 0.1 times the width of a grid cell), and check if the average error in the RBF's area of effect was made worse by this change. If it was, the change is reverted, and that dimension is excluded from further iterations. Otherwise, the change is kept, and the iterations continued. Intuitively, this algorithm can be viewed as blowing up the RBF like a balloon inside a closed box, with the sides of the box representing the widths at which further inflation along that axis begins increasing the error.

Alternatively, a fast, non-iterative method of selecting the RBF widths is Maximum Likelihood Estimation (MLE)[37]. In this method, the data to be fit is treated as a histogram of samples from a Gaussian probability density function with the previously computed mean. The width σ of the probability density function that would give this set of samples the highest probability of being generated is then computed. This computation can be performed independently for each dimension, giving $\sigma_{xi}, \sigma_{yi}, \sigma_{zi}$.

The equations for computing the MLE width for the x dimension are:

$$\begin{aligned} U_0 &= \sum f \\ U_x &= \sum xf \\ U_{xx} &= \sum (x - U_x/U_0) f \\ \sigma_{xi}^2 &= U_0/2U_{xx} \end{aligned} \tag{2.4}$$

where x is the x coordinate of the data point and f is the data value at that point. Other dimensions are computed similarly.

Since the residual data being fit can take on negative values, we must somehow convert this into a positive-valued histogram for the MLE computation. We do this by treating negative data values as zero when the RBF weight is positive, and the opposite when the RBF weight is negative (except then also taking the absolute value of the data values). The rationale for this is that we would like the RBF's region of influence to be restricted to an area in which the data has the same sign as the RBF, since these are the areas in which the RBF will decrease the fitting error rather than increase it. Setting values of the histogram to zero outside this area encourages probability distribution widths that do not produce many samples outside this area, and thus result in RBFs that do not have much influence outside this area.

A comparison between MLE width selection, iterative anisotropic width selection, and iterative isotropic width selection is given in table 2.1. The MLE method is clearly comparable in accuracy and yet is significantly faster to compute.

2.3.4 Weight selection

Once the RBF center and widths have been chosen, we compute the weights to minimize the sum of squared errors by forming the following system of linear equations, similar to [24, 15, 9]:

$$\begin{bmatrix} 1 & \phi_1(x_1) & \cdots & \phi_m(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(x_n) & \cdots & \phi_m(x_n) \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \tag{2.5}$$

where m is the number of RBFs, n is the number of data cells, x_j is the position of the jth data point in the volume, $f(x_j)$ is the data value at that point, w_0 is the constant term, w_i is the weight of the ith RBF, and ϕ_i is the ith RBF, with center μ_i and widths σ_i.

If $m = n$, then this system can be solved exactly. Typically we will have $m < n$, so in general we compute the best solution in the least-squares sense through the use of singular-value decomposition.

Data Set	Data Size	RBF Type	Width Selection	Fitting Time	Number of RBFs	RMS Error
UNC Head	256^3	Isotropic	Iterative	4.12 m	520 K	1.93%
UNC Head	256^3	Anisotropic	Iterative	9.48 m	455 K	1.50%
UNC Head	256^3	Anisotropic	MLE	2.25 m	485 K	1.75%
VHF Torso	512^3	Isotropic	Iterative	29.9 m	3.48 M	1.68%
VHF Torso	512^3	Anisotropic	Iterative	58.2 m	2.95 M	1.23%
VHF Torso	512^3	Anisotropic	MLE	19.1 m	3.22 M	1.60%
LLNL R-M	1024^3	Isotropic	Iterative	139 m	15.9 M	2.22%
LLNL R-M	1024^3	Anisotropic	Iterative	315 m	15.4 M	1.64%
LLNL R-M	1024^3	Anisotropic	MLE	85.3 m	16.0 M	1.85%

Table 2.1: Comparison of iterative fitting of isotropic RBFs, iterative fitting of anisotropic RBFs, and single-step fitting of anisotropic RBFs. See section 2.3.7 for a description of the data sets. The iterative fitting method can produce a somewhat better fit, but would take 3-4 times longer.

2.3.5 Choice of the Basis Function

Gaussian RBFs (as in equation 3.1) have seen much use in prior work [23, 26, 27, 24, 36, 9], largely due to the smooth blending that occurs when these RBFs are brought in close proximity to each other. Since we truncate RBFs at octree cell boundaries and perform explicit blending between the cells, we do not benefit from this natural blending. However, there are still some desirable properties of this basis function. First, it decays to zero rather than going off to infinity, so if an octree cell is poorly fit by its RBF in some region, there is a limit to how much damage the bad RBF can do to the final, blended result. Second, the derivatives of Gaussian RBFs do not degenerate at higher orders, so these basis functions are useful if higher-order derivative information needs to be computed. Still, most of our work is not dependent on the particular choice of basis function, and investigation into other possible basis functions may prove useful.

2.3.6 Binary Representation

Once the RBF representation has been computed, we store it in the following format. The nodes of the octree from section 2.3.1 are stored in breadth-first order. Each node contains exactly one RBF and has either eight or zero children.

We first store a 4-byte integer giving the byte offset to the start of this node's children. We then store the maximum and minimum data values contained within the node as two 2-byte integers. The maximum and minimum values are packed into 2 bytes each by subtracting their minimum value, dividing by their range, and then multiplying by the maximum size of a 2-byte integer. We next store the constant term w_0 and RBF weight w_1, each as a 4-byte float. We then store the x, y, z components of the RBF mean, each as a 1-byte integer. The packing computation takes into account the node's size and position in the octree. For each component we subtract the lowest value it could take on in that node, divide by the width of the node, and then multiply by the maximum size of a 1-byte integer. Finally, we store the $\sigma_x, \sigma_y, \sigma_z$ width factors, each as a 4-byte float. This gives a total size of 31 bytes per node, which we pad to 32 bytes so that it fits evenly into two 16-byte texture look-ups. This packing is illustrated in figure 2.4.

At the start of the file we store a header giving the number of levels in the octree and the minimum values and ranges of the node maximums and minimums, to allow the packing to be undone.

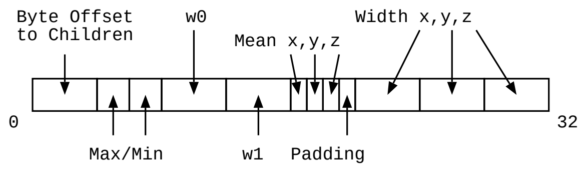


Figure 2.4: RBF and octree parameters packed into 32 bytes.

2.3.7 Fitting Results

We have tested our RBF fitting implementation on three different data sets. The first (UNC Head) is a CT scan of a human head from the University of North Carolina. This data set originally consisted of 113 slices, with each slice being a 256^2 array of 16-bit integers (although only 12 bits of precision are actually used). We linearly interpolated additional slices between the existing ones and duplicated the final slice several times to produce a data set of size 256^3. In our tests we used this data set as well as a version downsampled to 128^3.

Data Set	Data Size	File Size	Fitting Time	Number of RBFs	RBF File Size	RMS Error
UNC Head	128^3	4 MB	0.28 m	91.4 K	2.8 MB	2.22%
UNC Head	256^3	32 MB	1.90 m	485 K	15 MB	1.75%
VHF Torso	128^3	4 MB	0.38 m	109 K	3.4 MB	2.14%
VHF Torso	256^3	32 MB	2.10 m	520 K	16 MB	2.00%
VHF Torso	512^3	256 MB	16.2 m	3.22 M	99 MB	1.60%
LLNL R-M	128^3	2 MB	0.22 m	62.2 K	1.9 MB	5.24%
LLNL R-M	256^3	16 MB	1.52 m	402 K	13 MB	4.37%
LLNL R-M	512^3	128 MB	11.3 m	2.58 M	79 MB	3.31%
LLNL R-M	1024^3	1024 MB	85.3 m	16.0 M	489 MB	1.85%

Table 2.2: Fitting results for several different data sets using anositropic MLE-based RBF width selection. For comparison, the VHF Torso and LLNL R-M data sets have been downsampled to several different resolutions prior to fitting.

The second data set (VHF Torso) is a 512^3 block of the torso area of the Female CT data set from the Visible Human Project[38], in the same format as the UNC Head data. In our tests we used both the full 512^3 version and versions downsampled to 256^3 and 128^3.

The third data set (LLNL R-M) is time step 100 from the Richtmyer-Meshkov Instability data set from LLNL[5], which consists of an array of $2048 \times 2048 \times 1920$ 8-bit integers. We padded this to 2048^3 and then downsampled it to several different resolutions for testing.

We performed our fittings on a 64-bit Linux machine with dual Intel Xeon 3.0 GHz CPUs and 8 GB of RAM. As our fitting program is single threaded, it can only make use of one CPU.

Fitting results are given in table 2.2. For most of the data sets our fitting method achieves a reduction in file size of about 50% with an RMS error of about 1%-2%. An exception to this was the lower resolutions of the LLNL R-M data set. We suspect the reason for this is that these low resolution versions contain sharp, large-magnitude discontinuities which are not amenable to fitting with smooth Gaussian RBFs.

It is difficult to compare the efficiency of our fitting results with previous work due to a lack of prior reported fitting times. One exception to this is Hong *et al.* [26] who report fitting at the rate of 1300 grid points per minute. Although our results are several orders of magnitude faster, this is not a fair comparison since they are fitting unstructured data while we fit structured.

2.4 Rendering

We use ray-casting to render implicit RBF-encoded volumes. Data values at points in the volume are computed directly from the RBF representation without reconstructing the full data set in memory. Specifcally, the scalar value at a point is computed by summing up the values of all RBFs that overlap that point. The gradient vectors can be computed in a similar fashion, by summing the gradients of all RBFs that overlap the sample point (this works because the gradient operator distributes across summations). This sampling method can then be used to implement any rendering scheme that works by sampling data values and gradients at various points, such as direct volume rendering or point-based isosurface rendering.

2.4.1 GPU Direct Volume Rendering

We have implemented a GPU direct volume rendering algorithm using NVIDIA's new CUDA GPU programming system. Our implementation supports piecewise-linear transfer functions, gradient-based lighting, early ray termination based on accumulated opacity, and empty-space skipping based on checking whether the range of values contained within an octree cell overlaps the range which, given the transfer function, would produce any contribution to cumulative color or opacity. We also perform view-dependent level-of-detail by adjusting the ray step size and maximum octree depth based on the current distance along the ray.

Given an eye position and viewing direction, we construct ray origin and direction vectors for a perspective projection on the GPU and store them in GPU memory. We then execute a GPU kernel that traces each ray in parallel and writes the final resulting colors to GPU memory.

The kernel first sets the initial sample point to the ray's intersection with the data volume's bounding cube, and then begins stepping along the ray. At each sample point the kernel executes a space skipping routine that descends through the octree cells containing that point. For each cell, it makes a single 16-byte texture look-up and unpacks some parameters as described in section 2.3.6. It then checks whether the cell's range of data values would produce any contribution to the final color or opacity. If not, the kernel computes the ray's intersection with that octree cell and moves the sample point just beyond the intersection point. This is repeated until the skipping routine encounters a sample point at which it decends all the way through the octree.

The kernel then evaluates the scalar field and its gradient at the current sample point. This simply requires traversing the octree from the root to a leaf, evaluating the value and gradient of the RBF stored in each cell at the location of the sample point and summing up these values. We make use of the efficient location-code-based octree traversal methods of Frisken and Perry[39]. For each octree cell the kernel makes two 16-byte texture look-ups and unpacks some of the parameters as described in section 2.3.6. Finally, we sample color and opacity values from the transfer function using linear interpolation and apply diffuse, gradient-based lighting.

2.4.2 View-Dependent Level of Detail

Our multi-resolution RBF hierarchy allows us to implement view-dependent level-of-detail rendering by reducing the depth in the octree that we visit based

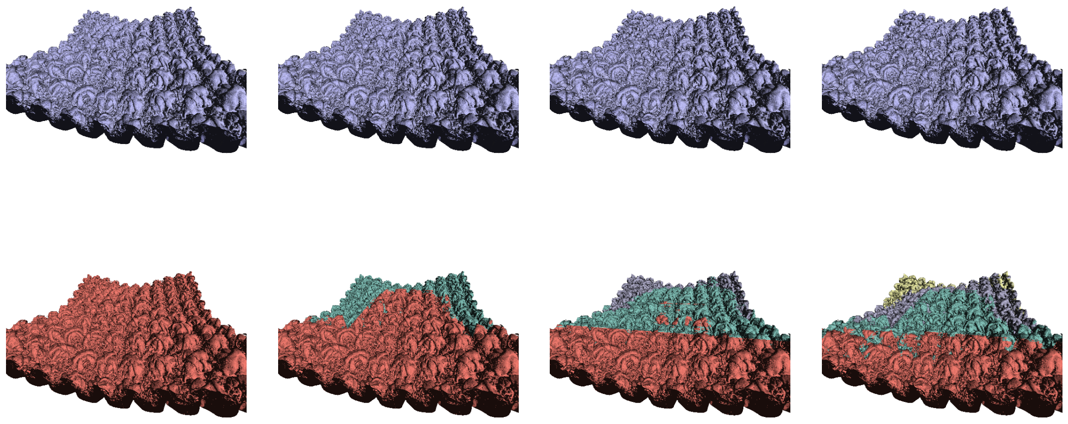


Figure 2.5: Comparison of varying amounts of view-dependent Level of Detail reduction (see section 2.4.2) for the 1024^3 LLNL R-M data set. The top row shows the renderings with the LOD reduction applied. The bottom row shows which LOD is being used in which area of the volume by color coding— from front to back, the colors represent levels of detail generated from 10, 9, 8, and 7 levels of the octree. From left to right, the rendering times are 0.59s, 0.53s, 0.48s, and 0.44s.

on the distance along the ray. Every time a certain amount of distance has been traversed, we reduce the maximum depth in the octree that we can visit by one level. To avoid level-of-detail transition discontinuities one can store the sample value computed at one level above the current level during the octree traversal, and blend between this sample and the final sample based on the distance from the previous level-of-detail transition. We experimented with this blending, but found that in practice the level-of-detail transitions tended to occur far enough away from the viewer that the discontinuities were not perceptible, even without blending. We did not use level-of-detail blending for the images and results in this chapter.

In addition to decreasing the octree depth based on the distance along the ray, we also double the ray step size every time we decrease the maximum depth by one level. We account for this during the incremental updating of cumulative color and opacity by keeping track of the integer ratio of the current step size to the original step size and iterating our incremental updates this many times at every step, using the same sample color and opacity values.

Screen shots of renderings using various amounts of view-dependent level-of-detail reduction and rendering rates are given in fig. 2.5.

2.4.3 Blending

Truncating the RBFs at octree cell boundaries can lead to visually noticeable discontinuities. To resolve this, we take into account data in adjacent cells when fitting each cell's RBF (see section 2.3.1). Then, for each sample point, in addition to computing a sample value from the RBFs in that point's octree cells, we also compute sample values at that position from the RBFs in adjacent octree cells. We implement this by simply altering the location code of the sample point to the location code of a point in each adjacent cell.

Since each cell's overlap region extends halfway into each adjacent cell, at each sample point there is the possibility of blending with up to three face-adjacent cells, three edge-adjacent cells, and one corner-adjacent cell. The contribution of each of the adjacent cells is linearly weighted based on the distance from the sample point to the corresponding face, edge, or corner (see fig. 2.3 for an example on a quadtree). This blending scheme is similar to the one used in Ohtake *et al.*'s multi-level partition of unity implicits[40].

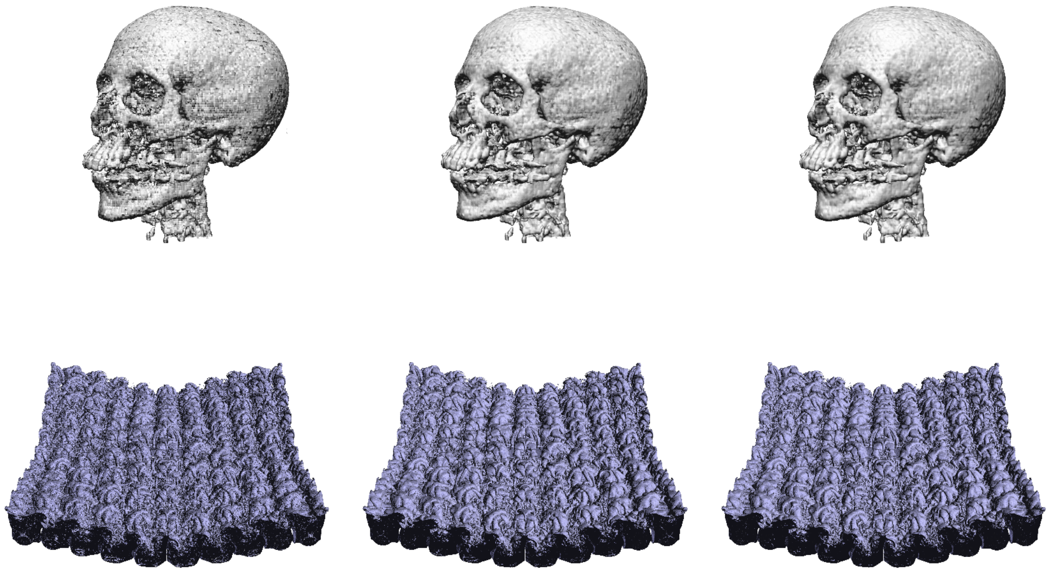


Figure 2.6: Comparison of (left to right) no blending, four-sample blending, and eight-sample blending (see section 2.4.3) for the 256^3 UNC Head and 1024^3 LLNL R-M data sets. Rendering times are given in table 2.3.

While completely smooth blending requires blending across faces, edges, and corners, many of the discontinuities can be resolved by simply blending across faces. This reduces the number of samples required per point from eight (1 local + 3 face + 3 edge + 1 corner) to four, reducing rendering time by about a factor of two. A comparison between using no blending, four-sample blending, and eight-sample blending is given in fig. 2.6 and table 2.3.

Data Set	Blending Type	Rendering Time
LLNL R-M	One-Sample	0.37 s
LLNL R-M	Four-Sample	0.72 s
LLNL R-M	Eight-Sample	1.11 s
UNC Head	One-Sample	0.10 s
UNC Head	Four-Sample	0.21 s
UNC Head	Eight-Sample	0.33 s

Table 2.3: Comparison of no blending, four-sample blending, and eight-sample blending (see section 2.4.3) for the 256^3 UNC Head data set and the 1024^3 LLNL R-M data set. Screen shots are given in fig. 2.6

2.4.4 Rendering Results

Unless stated otherwise, all renderings were done with MLE-fitted data (see section 2.3.3), four-sample blending (see section 2.4.3), and an image resolution of 512^2. Renderings were performed using a beta version of NVIDIA's CUDA drivers on a GeForce 8800 GTX with 768 MB of RAM. Rendering times for several different data sets and transfer functions are given in table 2.4, and screen shots are given in fig. 2.7.

We found that the ray step size required to avoid artifacts was dependent on

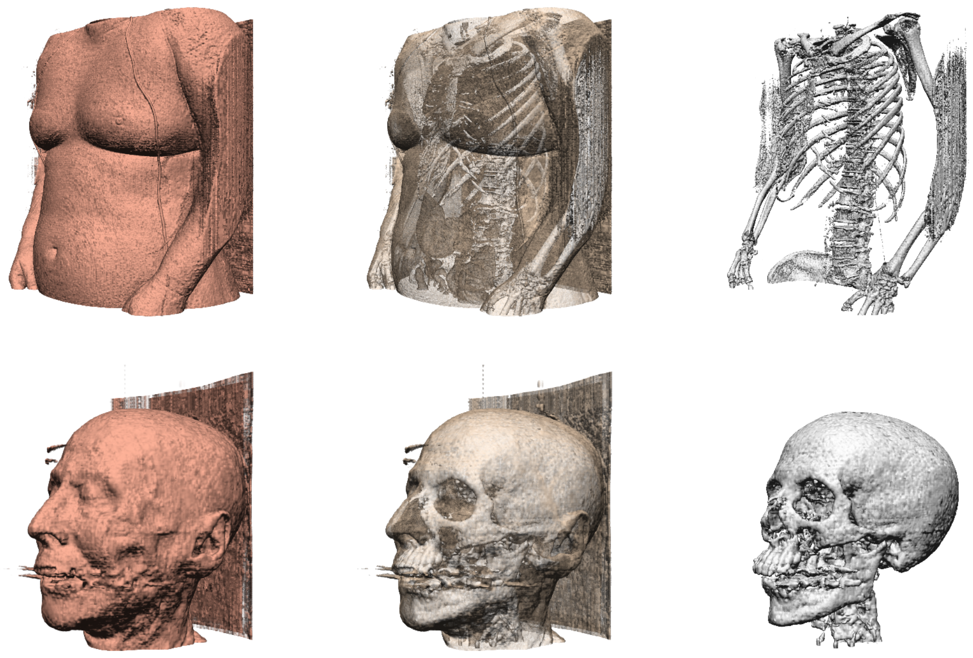


Figure 2.7: Rendering results using three different transfer functions for the 512^3 VHF Torso and the 256^3 UNC Head. Rendering times and other details are given in table 2.4.

the size of the smallest octree nodes in the fitting, which is related to the amount of high frequencies in the fitted data. We therefore needed to use smaller steps for higher resolution data. For a given data set and transfer function, most of the differences in rendering rate between original data resolutions are accounted for by the difference in step size.

2.5 Future Work

We plan to use this work on RBF fitting in the context of fitting scalar fields representing stem cells and nano fiber substrates. Our hope is that this will aid us in intelligently selecting an isosurface of this data set for use in later steps of our proposed algorithm.

A great deal of work has been done in the area of RBF fitting, for example in the machine-learning community. It seems likely that some of that work could further benefit procedural encoding through RBFs. Possible improvements include adding the ability to fit to a given error bound and removing the need to perform explicit blending between octree cells. It also might prove beneficial to investigate basis functions other than the Gaussian RBFs.

Another area of future work might be to investigate 4D fitting of time-varying data sets. Since these data sets often contain a large amount of temporal coherence, fitting multiple time steps simultaneously could result in a significant reduction in the number of basis functions required to represent the data set.

Finally, there are likely many possible uses of the higher-level information provided by the RBF representation that have not yet been explored, perhaps involving the use of high order derivatives that would have been too expensive to compute when using traditional volume rendering techniques.

	Transfer	Data		Rendering
Data Set	Function	Size	Step Size	Time
UNC Head	Skin	256^3	384^{-1}	0.45 s
UNC Head	Skin/Bones	256^3	384^{-1}	0.67 s
UNC Head	Bones	256^3	384^{-1}	0.21 s
VHF Torso	Skin	256^3	384^{-1}	0.32 s
VHF Torso	Skin/Bones	256^3	384^{-1}	1.09 s
VHF Torso	Bones	256^3	384^{-1}	0.30 s
VHF Torso	Skin	512^3	768^{-1}	0.54 s
VHF Torso	Skin/Bones	512^3	768^{-1}	2.15 s
VHF Torso	Bones	512^3	768^{-1}	0.53 s
LLNL R-M	Isosurface	256^3	384^{-1}	0.26 s
LLNL R-M	Isosurface	512^3	768^{-1}	0.40 s
LLNL R-M	Isosurface	1024^3	1536^{-1}	0.72 s

Table 2.4: Rendering results for several different transfer functions and data sets of several different sizes. All renderings were done using four-sample blending (see section 2.4.3). Step Size is the length of a step along the ray given that the data volume is a unit cube. Images of several of these data sets are given in fig. 2.7.

Chapter 3

Parallel Stochastic Surface Measurement

3.1 Introduction

In this chapter we discuss a parallel, stochastic algorithm to compute the surface area of a volumetric model of a molecule. The parallel nature of the algorithm makes it suitable for implementation on the GPU and makes it a good candidate for situations where high-throughput is required. The stochastic nature of the algorithm makes it progressive, allowing it to return a rough answer almost immediately and refine the answer over time to the desired level of accuracy.

We next describe the application of this algorithm to protein molecules modelled by Gaussian Radial Basis Functions, but the algorithm is general enough that it could be applied to any data representation that allows the computation of the intersections of a line with the object being measured. An application of some of the techniques used in this chapter to the problem of stem cell classification is given in chapter 5. The material in this chapter has been previously published [3].

Computation of molecular surface area is important in the grand challenge problems of molecular docking and protein folding as it allows one to incorporate the effects of solvent in the potential energy calculations. Recent work on interactive manipulation [41] and visualization of large-scale proteins [42] shows us how interactive visualization offers a powerful front end for computational steering of calculations. In such settings, rapid calculation of protein conformations becomes especially important and fast solvent-solute interactions are an essential first step. In this chapter we address the mapping of molecular surface area calculations on the emerging multi-core architectures for potential use in interleaved computation and visualization of large bio-macromolecular complexes.

To serve this need for molecular surface area computation, a wide variety of algorithms and programs have been developed— a few examples are the early works by Connolly [43] [44], MSMS [45] by Sanner *et al.*, GETAREA [46] by Fraczkiewicz and Braun, LSMS [47] by Can *et al.*, 3V [48] by Voss, and an adaptive grid-based algorithm [49] included in TexMol [50] by Bajaj *et al.*. These algorithms have been designed to work well on traditional, single-processor computer architectures using a serial programming model.

However, computer architectures are now facing the first major disruptive challenge in over two decades in the form of pervasive parallelism. For example, AMD and Intel have already changed their product lines to include dual-core and quad-core processors. According to the Intel road map, they plan to have hundreds of cores on a single chip becoming a reality over the next decade. The Cell processor has 8 stream processing cores in addition to a conventional scalar processor. GPUs have been at the forefront of the multi-core revolution in that they are already shipping with hundreds of cores. NVIDIA's G80 has 128 multiprocessors. Intel has recently disclosed their plans for a GPU consisting of 24-32 cores each involving a 16-wide SIMD vector processor with over 2 TFLOPs of performance. In addition, GPUs and CPUs are being merged thereby blurring the distinction between cores that specialize for graphics and cores that are more general-purpose. Both AMD and Intel are working on fused CPU-GPU cores; this will enable tight coupling between applications and graphics. Because of the large computer games market, these highly-parallel GPUs are being mass produced and are available for commodity prices. While the use of this hardware for scientific computation originally required some unpleasant hacks, recent development environments such as NVIDIA's CUDA (Compute Unified Device Architecture) [51] and ATI's CTM (Close To Metal) [52] make the use of this hardware much more elegant. Bringing GPU computing further into the mainstream is NVIDIA's Tesla product line, a GPU designed specifically for general-purpose computation.

Unfortunately, algorithms and programs designed for a single-processor architecture are often not able to directly take advantage of these new parallel processors. Algorithms designed for serial computation can sometimes be parallelized, but this can be a non-trivial task.

To take advantage of this new trend in high-performance computer architecture, we present a parallel algorithm, implemented for both CPU and GPU, to efficiently compute molecular surface area. In addition to its parallel nature, the algorithm is also progressive, providing a rough estimate of surface area very quickly and refining the estimate over time until the desired accuracy is reached. Finally, the algorithm generates points on the molecular surface, which can be used to create point-based renderings of the molecule.

3.2 Related Work

3.2.1 Molecular Surface Area Computation

Molecular surface areas have been computed through several different methods. The program MSMS [45] by Sanner constructs the Solvent Accessible Surface (SAS) [53] and Solvent Excluded Surface (SES) [54] by considering the intersections of spheres representing Van der Waals radii of atoms of the molecule, and using this information to compute a set of patches which make up the surface. The reduced surfaces it computes correspond to alpha shapes [55]. The program GETAREA [46] by Fraczkiewicz and Braun also calculates surface area by computing surface patches based on sphere intersections, making use of some additional ideas from computational geometry. A different type of approach was used by Wodak and Janin[56], who give a fast method to estimate molecular surface area using only distances between pairs of atoms.

Additionally, any program that computes triangulations of molecular surfaces, such as SURF [57] by Varshney *et al.*, can be easily converted to give an estimate of molecular surface area by adding up the areas of all the generated triangles. SURF is designed to take advantage of data-parallelism at the granularity of individual atoms, but cannot scale to take advantage of an unlimited degree of parallelism as our algorithm can. The SURF algorithm is also restricted to molecules defined as a collection of discrete atoms, while our algorithm can be applied to molecular surfaces defined in virtually any manner.

More recently, the program LSMS [47] by Can *et al.* discretizes atomic Van der

Waals spheres onto a regular grid, and then uses the level-set method to propagate fronts to compute the SAS and SES; it can also compute the Solvent Excluded Volume (SEV). The program 3V [48] by Voss also discretizes the molecule onto a regular grid and computes area and volume, but does not use the efficient level-set algorithm of LSMS.

Another recent algorithm by Bajaj and Siddavanahalli [49] can compute several different molecular surfaces. Their work models atoms using signed distance fields, which are similar to the radial basis functions used in our work. However, our algorithms are very different— Bajaj and Siddavanahalli's algorithm builds up molecular surfaces incrementally on a grid by adding atoms one at a time, while our algorithm measures surface area using parallel, stochastic sampling.

3.2.2 General Purpose GPU Computing

Although Graphics Processing Units (GPUs) were originally specialized hardware suitable only for 3D graphics computations, modern GPUs have evolved into general-purpose high-performance parallel processors. NVIDIA's G80 product line, for example, features 128 programmable processor cores and advertises a maximum performance of 300 gigaflops. These processors are programmed in an SPMD (Single Program, Multiple Datastream) fashion; all processors execute the same program, but are allowed to take different branches at conditional statements at the cost of a performance penalty. The high peak performance of GPUs relative to CPUs is largely due to the fact that GPUs devote a larger proportion of their transistors to arithmetic computation instead of tasks such as memory caching. Because of this architecture, GPUs perform best with algorithms that do a large amount of computation relative to their number of memory accesses; this type of algorithm is referred to as having high arithmetic intensity.

Modern GPUs have large amounts of on-card memory; first generation Tesla cards, for example, will have 1.5 GB of RAM. Historically, each processor on a GPU was only able to write its output to a single location in memory, corresponding to the pixel whose value that processor was computing. Modern GPUs have overcome this limitation and allow full read and write access to any location in memory from any processor. Additionally, the processors have access to a small pool of very fast shared memory which is suitable for communication between processors within the inner loop of an algorithm.

In the past, writing a general-purpose program for a GPU meant casting the algorithm in terms of graphics operations, such as texture look-ups and RGB color vector manipulations. With the recent advent of development environments such as NVIDIA's CUDA and ATI's CTM, however, general-purpose algorithms can be written in much more natural terms. CUDA, for example, is basically equivalent to the C language with a few extensions to facilitate the launching of parallel computation kernels.

Even though GPU algorithms can now be written in development environments similar to those used for CPU algorithms, developing an algorithm for a highly parallel architecture such as a GPU requires a different approach than developing for current CPUs. For an algorithm to run efficiently on a GPU, it must be divided into a large number (at least on the order of hundreds) of independent tasks which can be executed simultaneously. We note that our algorithm is ideal for this, since it is based on taking a large number of independent random samples. Care must also be taken to reduce main memory access as much as possible, and to take advantage of the available fast shared memory.

3.3 Gaussian Molecular Modelling

We calculate the surface area of a protein which is represented as the level set of a sum of Gaussian Radial Basis Functions (RBFs), with one RBF being placed at the location of each atom's center. This implicit molecular surface representation has been used as far back as Blinn's 1982 work [58], as well as in many more recent works such as Grant and Pickup[59], Ritchie[60], and Bajaj and Siddavanahalli [49].

Symbolically, each Gaussian RBF $\phi(x)$ can be represented as

$$\phi_i(x) = w_i \; e^{-\| \frac{x-\mu_i}{\sigma_i} \|^2} \tag{3.1}$$

where w_i is the weight of the ith RBF, μ_i is the location of the ith RBF's center, and σ_i controls the width of the ith RBF.

The program reads the same XYZR file format used by MSMS [45], which can be generated from PDB [61] files by the *pdb_to_xyzr* utility that comes with MSMS. Note that multiple atoms may be combined into a single entry in the PDB file (merging with hydrogens, for example), in which case the number of RBFs will be different than the number of atoms in the protein.

For the results reported in this work, we set the μ_i to the RBF centers in the XYZR file. RBF weights and widths are set based on the constants given in Ritchie[60] and Grant and Pickup[59], which are designed to model the Van der Waals surface of a molecule. Specifically, we set $w_i = 2.70$ for all RBFs and $\sigma_i = r_i/\sqrt{2.3442}$, where r_i is the RBF radius from the XYZR file. We form the overall scalar field by summing together all RBFs, and treat the surface as being at an isovalue of 0.259.

To accelerate sampling of the scalar field, we insert the RBFs into a bucketing spatial data structure. We partition space into a regular grid, and store a pointer to a list of the RBFs that overlap each grid cell at the corresponding element of a three-dimensional array. The Gaussian RBFs are truncated to zero at a radius of 3σ.

3.4 Stochastic Area Measurement

To measure molecular surface area we make use of the Cauchy-Crofton formula (equation 3.2) from integral geometry, which relates the area of a surface to the number of intersections with the surface of a set of lines. This formula can be written as

$$\int m \, dL = \pi s \tag{3.2}$$

where s is the surface area, m represents the number of intersections along a given line, and the integration is taken over the space of all possible lines.

A numeric approximation to this integral can be made by taking a random sample of lines and counting their intersections with the surface. Approximating the integral in this manner gives $\frac{n_1}{N} \approx c\pi s_1$, where n_1 is the count of intersections, N is the number of sampled lines, s_1 is the surface area, and c is an unknown constant of proportionality. To get rid of c, we can intersect the same set of lines with a second surface, giving $\frac{n_2}{N} \approx c\pi s_2$. Combining these equations gives

$$s_1 \approx \frac{n_1}{n_2} s_2 \tag{3.3}$$

If the area of the second surface s_2 is known, we can then calculate the molecular surface area s_1. This derivation is given in more detail in Li *et al.*[62], and further applications are discussed in Liu *et al.*[63].

3.4.1 Sampling the Space of Lines

Several methods for generating lines from randomly chosen parameters are given in Li *et al.*[62]. We use a method called the Chord Model, which consists of picking two random points from a uniform distribution of points on the surface of a sphere and then taking the line that passes through them. Uniformly distributed points (x, y, z) on a sphere can be generated from pairs (u, θ) of uniformly distributed random numbers by using the formula

$$(x, y, z) = ((1-u^2)^{\frac{1}{2}} \cos\theta,\ (1-u^2)^{\frac{1}{2}} \sin\theta,\ u) \tag{3.4}$$

where u is in $[-1, 1]$ and θ is in $[0, 2\pi)$. Further discussion of generating uniformly distributed points on spheres is given on the Mathworld web site [64].

To generate random lines with this method, we must generate uniformly distributed random numbers. This would typically be done using a pseudo-random sequence; however, better results can be obtained by using a quasi-random sequence (also called low-discrepancy sequences). These sequences have less clustering of values than pseudo-random sequences, which results in a more representative sampling of lines and actually provides an asymptotically lower error bound for the numeric integration [62].

In our implementation we used the Niederreiter quasi-random sequence [1], which can be found in the GNU Scientific Library [65]. We generate 4D quasi-random points (a, b, c, d), and use the first and second coordinate pairs (a, b) and (c, d) to generate the (u_1, θ_1) and (u_2, θ_2) for equation 5.1.

Comparisons between 2D points generated from a pseudo-random, a quasi-random, and a regular grid distribution are given in figure 3.1. Note that the pseudo-random distribution has more clusters and bare regions than the quasi-random distribution. The regular grid distribution also avoids clustering, but is so regular that its use could cause aliasing artifacts. Further analysis of sampling points using quasi-random distributions can be found in Rovira *et al.* [66].

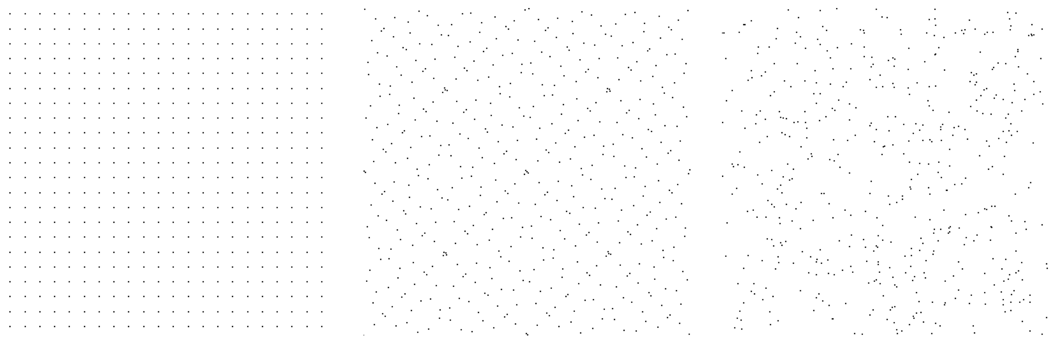


Figure 3.1: 2D distributions of points generated on a regular grid (left), from the Niederreiter quasi-random sequence [1] (middle), and from a pseudo-random sequence (right).

3.4.2 Intersection Counting

For our line intersection algorithm, we start by enclosing the RBFs representing the atoms in the tightest bounding sphere centered at the center of the molecule. One optimization could be to instead use the tightest bounding sphere, computed by a method such as in Gartner [67].

We then generate a sequence of quasi-random lines using pairs of points on the surface of the bounding sphere as described above. For each line, we step in uniform increments from one point to the other, evaluating the scalar field at each step to determine whether the current point lies in the interior or exterior of the surface. The optimal step size is a function of the typical atomic radii and packing densities. In this work we have used a step size of 0.25 Angstroms, which we have experimentally determined to be a reasonable value.

To evaluate the scalar field at a point, we iterate over all RBFs that overlap that point's bucket, adding their values to a running total until either all RBFs have been processed or the current total exceeds the surface's isovalue. If a point is found to be in the interior of the surface and the previous point was in the exterior (or vice versa), a running count of surface intersections is incremented.

Once the number of intersections of the lines with the isosurface has been computed, equation 3.3 can be used to estimate the surface area of the molecule (the area of the bounding sphere can be easily computed analytically, and the number of intersections with the bounding sphere is simply two times the number of lines intersected). The approximation improves as more lines are intersected.

3.5 Parallelization for GPU

Because the sampling along each line is completely independent from all other lines, this algorithm is a natural fit for a highly parallel architecture such as a GPU. In fact, our algorithm is able to linearly scale to take advantage of an unlimited amount of parallelism, since each additional available processor can be assigned to compute the intersections of another random line, increasing the speed at which the result converges.

We have implemented a version of the algorithm in NVIDIA's CUDA language that runs on a GPU, and compared its performance to the CPU version. In our GPU implementation the 4D quasi-random points that define the sample lines are generated on the CPU and then sent to GPU memory. After the per-line intersection counts are computed in parallel on the GPU, this data is sent back to CPU memory where the per-line counts are aggregated into an overall total. This process could potentially be optimized by computing the quasi-random points and performing the summation of the per-line counts on the GPU, which would not only take further advantage of the GPU's parallel processing capabilities but also avoid time-consuming data transfers to and from the GPU.

3.6 Test Results

Areas can be calculated for several different types of molecular surfaces. The Van der Waals surface is formed by a union of spheres located at the centers of the molecule's atoms, with radii equal to the atoms' Van der Waals radii. The

Solvent Accessible Surface (SAS) [53] is defined as the surface traced by the center of a probe sphere (representing a solvent molecule) as it is rolled along the Van der Waals surface. Finally, the Solvent Excluded Surface (SES) [54] is the boundary of the area that no part of such a probe sphere may penetrate. Programs can also calculate either the area of the outermost shell of the surface only, or include the area of any interior cavities as well.

For our tests, we set the parameters of our Gaussian RBF implicit surface representation to approximate the SES formed with a probe radius of 1.4 Angstroms. Our algorithm calculates the surface area of the outer surface as well as the interior cavities. We compare our results against several other programs that compute molecular surface area— MSMS [45], LSMS [47], and SURF [57].

Figure 3.2 illustrates how our algorithm converges on an estimate of the SES for the several proteins as increasingly more lines are intersected with the surface. In the remainder of the tests we set the number of intersected lines to 20,000, which we have found gives quick estimates with reasonable accuracy. The number of sample lines could be set higher or lower based on the speed and accuracy requirements of a particular application.

To evaluate the accuracy of our surface area computations, we would like to have some ground truth to compare our results against. Both MSMS and SURF compute the SES analytically, and their reported surface areas usually agree very closely. Therefore, we take the true SES area to be the average of the SES areas reported by MSMS and SURF, and measure algorithm accuracy as a percent difference from this average value. The differences between the area we report and

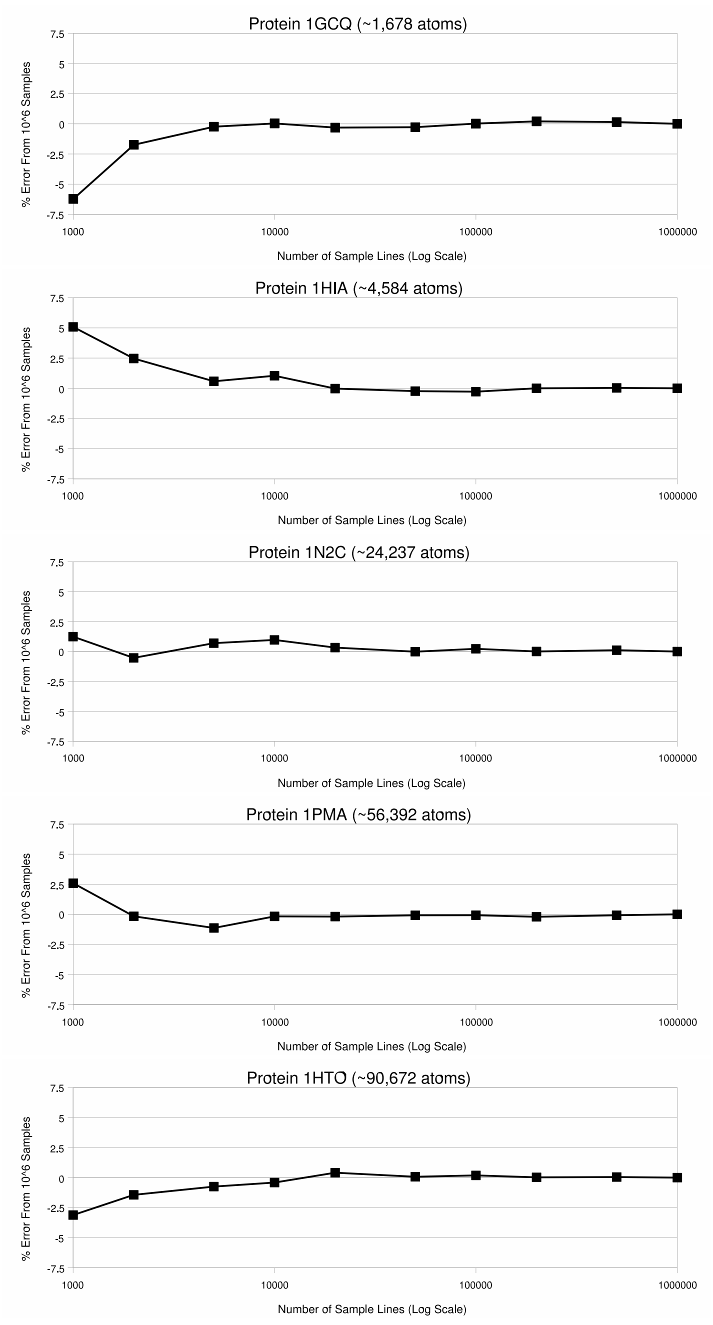


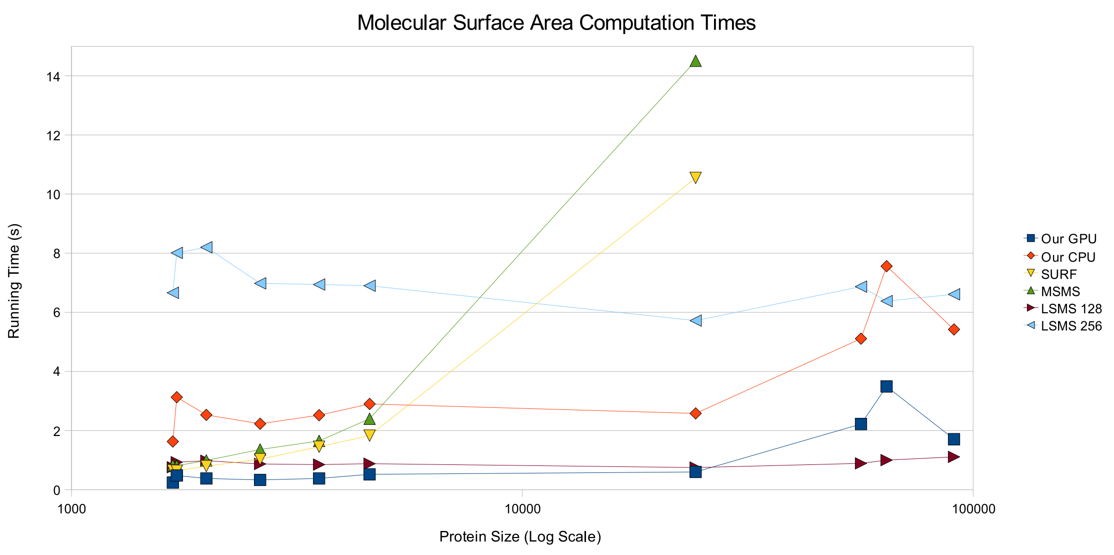
Figure 3.2: Surface area approximation errors for several proteins using various numbers of intersected lines. Vertical axis is the percent error from the final value (the area returned with 10^6 sample lines). Note that all proteins have converged to near their final area by 20,000 lines.

this average area come from two main sources— our algorithm not having yet fully converged on the area of our surface, and the fact that the surface whose area we are converging on is not quite the same as the SES surface that we are comparing ourselves against.

As can be seen from table 3.6, our differences are comparable to the differences of LSMS when using a fine 256^3 grid, while our GPU running time is significantly faster than LSMS using a 256^3 grid, and is often even faster than LSMS using a coarse 128^3 grid. We observe that our running time depends mostly on the molecule size, while for LSMS the running time depends mostly on the grid resolution. Our GPU implementation is also faster than MSMS and SURF, especially for larger molecules. A graph of running times of the various algorithms is given in figure 3.3. Note that for similar error levels our GPU algorithm is 3x – 10x faster than existing CPU algorithms.

3.7 Discussion and Future Work

Because our algorithm generates points on the molecular surface, it can easily be used to create point-based renderings [68] of the molecule. Surface normals for lighting calculations can also be easily generated by analytically computing the gradient of the implicit function at each surface point. Because of their light weight and simplicity, points are a good primitive for the representation of large models. Some point-based molecular renderings generated from our implementation using different numbers of points are shown in figure 3.4.



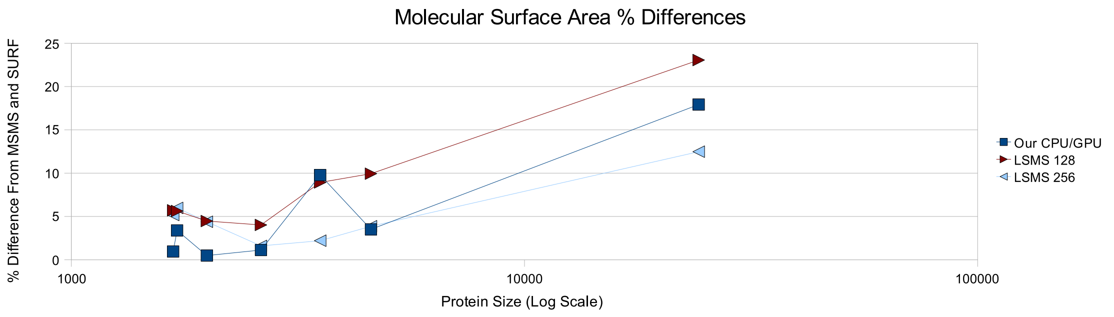


Figure 3.3: Surface area computation times and percent differences for proteins of different sizes using several different algorithms. Data is from table 3.6. Protein size is given in RBFs, which is equal to the number of atoms listed in the PDB file. For MSMS and SURF, data is not available for the larger proteins since these programs were unable to compute the area for proteins of that size. Percent differences are given as absolute magnitudes.

One nice feature of our algorithm that we have not explored is its progressive nature. As the algorithm runs, a rough approximation of the surface area is returned almost immediately, while increasingly accurate approximations are obtained as more and more line intersections are computed. This feature could be used to

Protein	1GCQ	2PTN	1PPE	8TLN	2CHA	1HIA	1N2C	1PMA	1FFK	1HTO
Size	1,678	1,712	1,991	2,621	3,542	4,584	24,237	56,392	64,268	90,672
Our Area (Å^2)	8,806.06	8,258.64	8,805.47	11,256.8	16,761.7	21,157.2	79,010.5	192,382	461,631	335,624
Area Diff	0.95 %	3.39 %	0.48 %	1.12 %	9.78 %	3.51 %	17.92 %	—	—	—
CPU Time (s)	1.63	3.13	2.53	2.23	2.52	2.90	2.58	5.11	7.56	5.42
GPU Time (s)	0.24	0.48	0.38	0.33	0.38	0.52	0.60	2.22	3.49	1.71
LSMS 128^3 Area (Å^2)	8,225.56	8,437.62	9,155.30	10,926.5	16,919.5	19,748.0	74,059.3	184,464	422,295	301,674
Area Diff	5.71 %	5.63 %	4.47 %	4.02 %	8.93 %	9.94 %	23.06 %	—	—	—
Time (s)	0.76	0.93	0.98	0.87	0.85	0.88	0.75	0.89	1.00	1.11
LSMS 256^3 Area (Å^2)	8,272.37	8,466.53	9,148.43	11,202.1	18,171.1	21,078.2	84,250.1	203,324	472,889	352,981
Area Diff	5.17 %	5.99 %	4.39 %	1.60 %	2.20 %	3.87 %	12.48 %	—	—	—
Time (s)	6.66	8.01	8.20	6.98	6.94	6.90	5.72	6.87	6.38	6.61
MSMS Area (Å^2)	8,724.65	8,039.77	8,807.21	11,364.4	18,538.6	21,944.8	97,129.4	—	—	—
Time (s)	0.83	0.81	0.99	1.36	1.65	2.40	14.51	—	—	—
SURF Area (Å^2)	8,722.10	7,935.88	8,719.81	11,404.0	18,619.9	21,909.9	95,388.7	—	—	—
Time (s)	0.66	0.64	0.80	1.03	1.45	1.83	10.55	—	—	—

Table 3.1: Comparison of the surface areas and running times of our method, LSMS with a 128^3 grid, LSMS with a 256^3 grid, MSMS, and SURF. Protein size is given in RBFs, which is equal to the number of atoms listed in the PDB file. For our method, we used 20,000 sample lines. All methods computed the SES area (or an approximation to it) of all disconnected surface components using a probe sphere radius of 1.4 Å. Tests were performed on a machine with a GeForce 8800 GTX GPU, an Intel Xeon 3.0 GHz CPU, and 4 GB of RAM. A '—' means that MSMS or SURF was unable to compute a surface for this molecule.

tune the speed versus accuracy of the algorithm for different applications, or to provide a rough estimate to decide whether or not more exact calculations are worth performing.

One possible area of future work might be to extend the program to compute other geometric properties of molecules, such as volume or mean curvature, as dis-

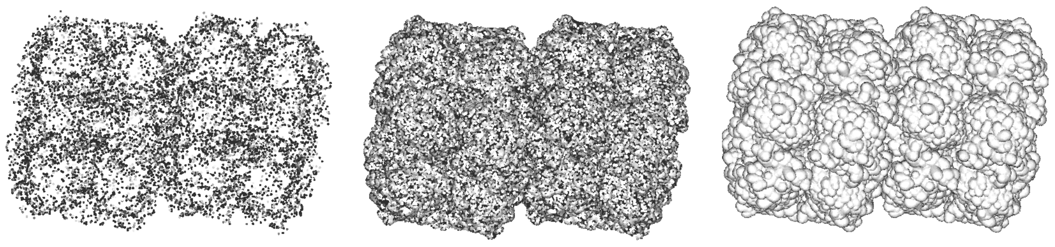


Figure 3.4: Point-based renderings of the protein with Protein Data Bank ID 1HTO using 10^4 points (left), 10^5 points (middle), and 10^6 points (right).

cussed in Schröder [69]. Molecular volume computation in particular would likely be an easy and useful extension.

One final issue worth mentioning is the treatment of hollow cavities within the interior of a molecule. Depending on the application, it may or may not be desirable to include interior cavity surface area in the overall surface area reported. A discussion of these interior cavities can be found in Liang *et al.* [70]. Our algorithm includes the surface area of these cavities in the final figures reported, as does SURF. MSMS and LSMS give an option to either include these areas or not. All surface areas and running times reported in this work are for the outer surface plus all cavities. If only the outer surface area is required, MSMS can compute this several times faster than it can compute the area of the outer surface plus all cavities.

Chapter 4

Confluent Visualization

In this chapter we discuss the visualization of the interaction between a pair of data sets. For example, the phenomenon of stem cell differentiation being induced by surface geometry involves a data set of the cell itself and a data set of the surface on which it is resting. Biologists may wish to visualize the area of contact between the cell and the surface to learn about how the surface induces differentiation. The idea of choosing an area of a data set to visualize based on another data set is called Confluent Visualization.

In the following material in this chapter we discuss Confluent Visualization in the context of gyrokinetic plasma turbulence simulations. However, the idea should be applicable to any pair of volumetric data sets whose interaction needs to be visualized. The material in this chapter was previously published [4].

Gyrokinetic simulations of tokamak turbulence are widely used to interpret experimental data. The level of realism in modern gyrokinetic codes is very high – so high, in fact, that it is nearly as difficult to understand and interpret the simulation results as it is to work directly with experimental data. One of the main problems is that the datasets, such as those shown here, are typically of dimension greater than three. Also, many gyrokinetic simulations take place in a flux-tube following coordinates whose geometry is fairly non-trivial and whose size with respect to the enveloping tokamak device is small. Moreover, the very nature of turbulence implies the existence of structures at various scales, which turns coherent graphical representation of data into a challenging problem.

A gyrokinetic simulation describes the time evolution of the five-dimensional probability distribution function(s) for a few plasma species (typically 1-5). Three of these dimensions are spatial and the other two correspond to velocity space variables. Visualizing such a dataset directly is clearly impossible. A traditional diagnostic quantity is the electric potential ϕ whose fluctuations capture the patterns of turbulence present in the system.

One of the most compelling techniques for visualizing three dimensional scalar fields is direct volume rendering. It gives a global qualitative view of the entire dataset, providing a fast insight into spatial patterns and correlations. However, direct volume rendering works best for scalar fields with high degree of regularity, such that occlusion and cluttering can be eliminated by filtering easily identifiable value ranges. This is not the case with the electric potential, or other turbulent scalar fields that arise in this context, which exhibit fluctuations on a variety of scales.

To deal with this problem we focus on derived diagnostic quantities, which are both physically meaningful and suitable for direct volume rendering. One such diagnostic is the heat flux, which is calculated as a velocity space integral:

$$Q = \int f E \frac{\partial \langle \phi \rangle}{\partial y} \, d\mathbf{v}$$

where f is the gyrokinetic probability distribution function, E is energy, and

$\langle \phi \rangle$ denotes the gyroaveraged electric potential. The values of Q are used to identify regions where the heat flux satisfies a certain condition, for instance, specified by a given range. Within each region of interest we map the values of a turbulent field, such as the electric potential. Thus information from two fields with distinct spatial characteristics is convolved into a single coherent visual representation.

The data presented here are taken from an electrostatic, gyrokinetic simulation of the Cyclone [71] benchmark case for ion temperature gradient-driven turbulence in a tokamak. The spatial simulation domain is $(n_x, n_y, n_z) = (96, 96, 64)$, where n_x and n_y are the number of grid points in the plane locally perpendicular to the background magnetic field, and n_z is the number of grid points along the magnetic field. There are 768 velocity-space grid points at every spatial grid, for a total meshpoint count of about half a billion. As is standard in the Cyclone benchmark, there is one gyrokinetic ion species, and the electrons are taken to have a Boltzmann response. The code used to generate this data is the GS2 code [72], [73].

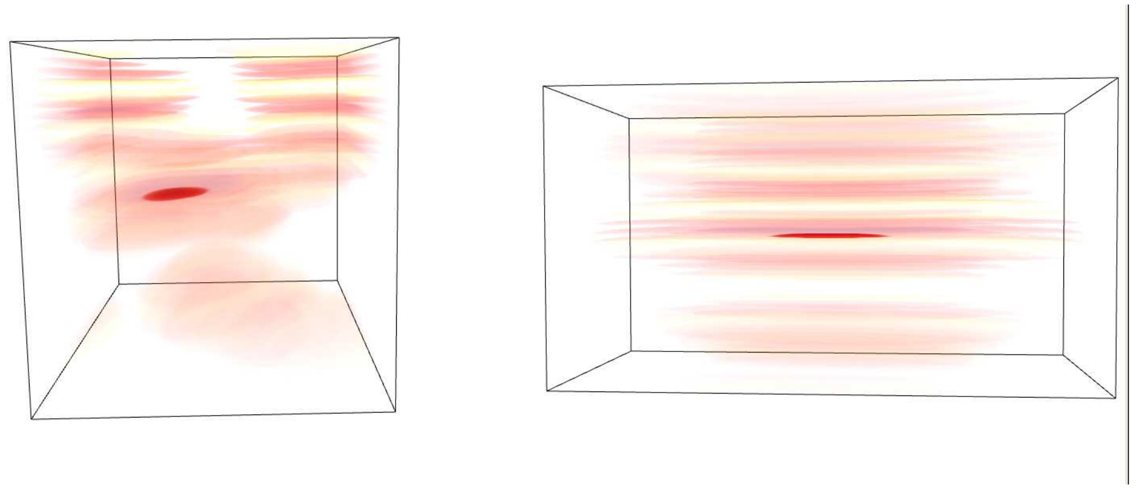


Figure 4.1: Computational Domain: Visible Structures Indicate Zones of High Heat Flux. Color Represents Values of the Electric Potential.

Figure 4.1 shows ”confluent” volumetric rendering (CVR) of the heat flux and the electric potential in the computational domain. Heat flux values are used in the opacity transfer function, in effect defining a set of visible spatial structures inside the volume. Electric potential values are used in the color transfer function, which controls how color is applied to visible pixels. In our setup we are looking at a collection of island regions where heat flux is higher than a certain cutoff value. Figure 4.2 illustrates the geometry of the flux tube domain and features a blown-up region that corresponds to a small ball around the center of the computational domain. Note that for aesthetics reasons the partial toroidal shell shown as a reference here is taken from a different tokamak configuration.

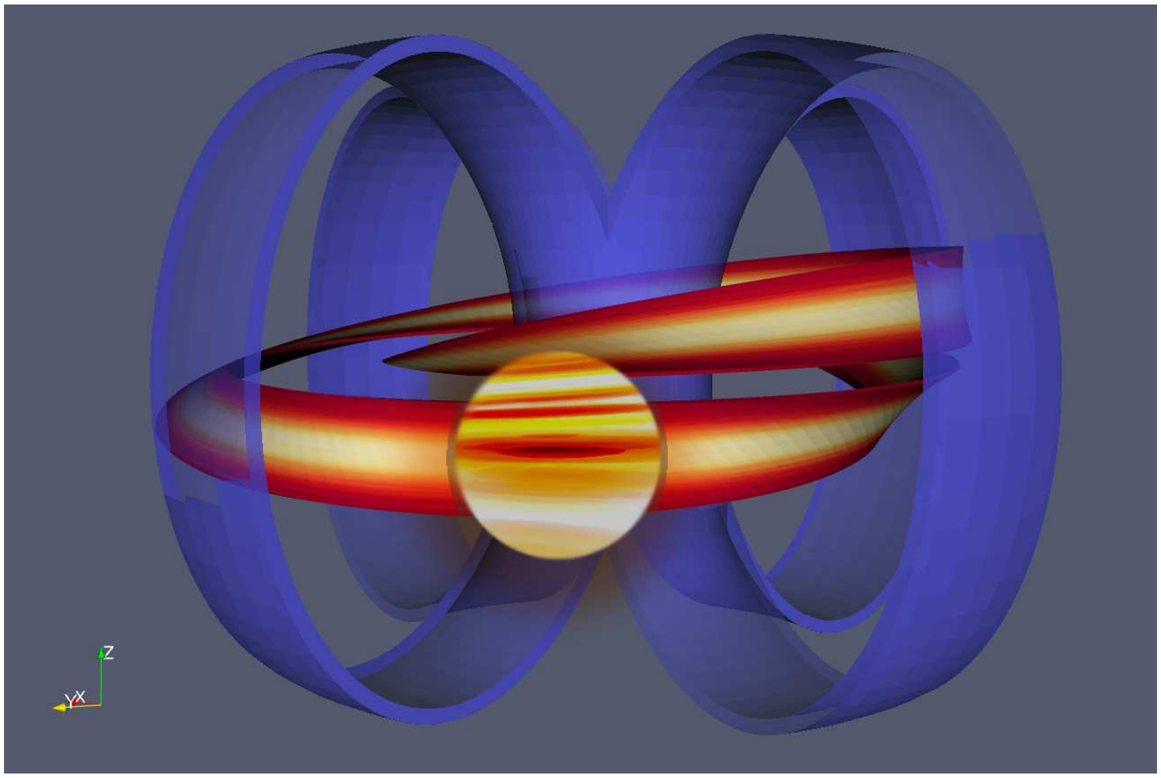


Figure 4.2: Flux Tube Domain in a Toroidal Device. The Circular Region is an Enlargement of an Area of the Ribbon Shaped Domain to Show Detail.

Shown here are data derived from the late linear phase of the simulation, when the linearly unstable modes are at high amplitude and are strongly interacting, but before the nonlinearly generated flows and eddies have reached high amplitude. The amount of heat being transported at this instance is quite large (comparable to the steady-state, turbulent value), despite the small spatial filling factor. A conventional diagnostic would show ”streamers” at this point in time. We are using the new CVR diagnostic to develop intuition about the relationship between the $E \times B$ flows (along contours of constant potential) and the radial heat flux.

Chapter 5

Parallel Stochastic Classification of Stem Cells

5.1 Introduction

In this chapter we discuss a parallel, stochastic algorithm that can classify volumetric models of stem cells as having been grown on a surface that induces cell differentiation or on a surface that does not induce cell differentiation. This algorithm builds on our algorithm discussed in Chapter 3 for molecular surface area measurement. Like the surface area measurement algorithm, this algorithm is suitable for implementation on a GPU and is progressive, allowing it to return a rough classification almost immediately and refine the accuracy of the classification over time.

In the following material in this chapter we discuss the application of this algorithm to the classification of stem cells, but the algorithm could be applied to the classification of any volumetric data sets that can be characterized by the lengths of the intersections of randomly generated lines with the data set. The material in this chapter is under preparation for publication [74].

Tissue engineering has been defined as an "interdisciplinary field that applies the principles of engineering and the life sciences toward the development of biological substitutes that restore, maintain, or improve tissue or whole organ function" [75]. Recent advances include many tissues and organs, including heart[76], bladder[77], and urethra[78].

The two critical parts of a tissue-engineered implant are stem cells and 3D tissue scaffolds [79]. Stem cells have the ability to differentiate down multiple lineages for regeneration of different organs and tissues. Scaffolds provide a 3D template for stem cells to adhere and differentiate.

Traditionally, control of cell function has been considered in terms of soluble factors, biochemical signaling and paracrine effects. However, recent work shows that the physical properties of the cellular microenvironment can also influence cell function. In particular, the chemistry [80, 81, 82], mechanics [83, 84], and structure [85, 86, 87] of the cell niche are important. Thus, a primary goal in regenerative medicine is to engineer scaffolds to provide a 3D microenvironment that enhances tissue regeneration.

Previously, we demonstrated that 3D PCL (poly(ε-caprolactone)) nanofiber scaffolds drive hBMSCs (human Bone Marrow Stromal Cells) into an elongated and highly-branched morphology that induces them to differentiate down an osteogenic lineage [86]. hBMSCs are isolated from bone marrow and contain adult skeletal stem cells that can differentiate into bone, fat and cartilage [88]. When hBMSCs were cultured on 2D flat PCL films (spun-coat), they assumed a well-spread, polygonal morphology, that supported cell proliferation but did not induce osteogenic differentiation. These results indicated that the structure of the scaffold can be designed to drive cells into morphologies that direct their differentiation down a desired lineage. Images of the 2D and 3D surfaces and cells are given in figure 5.1.

The use of scaffold structure to control stem cell function is attractive because

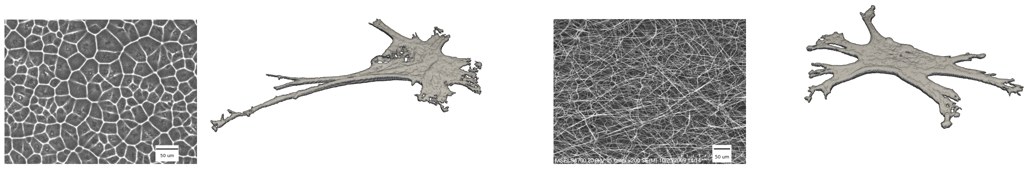


Figure 5.1: Left to right: SEM (Scanning Electron Micrograph) of 2D PCL spun-coat film, an hBMSC cultured on a 2D PCL spun-coat film, SEM of 3D PCL nanofiber scaffold, and an hBMSC cultured in a 3D PCL nanofiber scaffold. hBMSC images are reconstructed from 3D confocal fluorescent scans of fluorescently stained actin.

scaffold structure is stable, has a low regulatory burden and is relatively easy and inexpensive to control [89]. Covalent functionalization of scaffold devices with biochemically active molecules is difficult, hard to characterize and expensive. Loading scaffolds with growth factors to guide regeneration is challenging because proteins are hard to manufacture and highly unstable. In addition, including biomolecules or growth factors in a device increases the regulatory costs.

Cell shape and function are known to be intricately linked [90, 91], and recent work has shown that this premise holds true for hBMSCs [86, 87, 92, 93]. Cell shape has the added value of being an early predictor of cell fate. Biochemical assays require weeks or months of culture for differentiation markers to become detectable, whereas cells attain a stable morphology within a day of culture that can be an indicator for their future behavior [93]. Though previous methods for assessing cell shape have focused on 2D cell shape data [94, 93], tissue engineering aims to use 3D scaffolds to control cell shape function. Culture of cells in complex

3D microenvironments is likely to require 3D cell shape analysis in order to establish meaningful relationships. Rapid throughput is also desirable due to the large number of parameters that must be tested to identify scaffolds that promote the desired biologic response [95, 96].

In addition to the tissue engineering field, rapid 3D methods for measuring cell shape are likely to be needed by the pharmaceutical industry as it moves towards 3D scaffold systems for drug screening [97, 98]. 3D *in vitro* culture models are less expensive than animal models and may be more predictive of human clinical outcomes. Drug screens typically involve thousands of compounds and cell shape analysis is a parameter frequently used to determine toxicity response. Thus, high-content 3D methods for cell shape classification are required if pharma is to use 3D scaffold technologies to improve the predictive nature of *in vitro* testing [99, 100]. Herein, we have developed a machine learning algorithm that can rapidly identify differentiating stem cells based on their 3D shape.

The remainder of this chapter is organized as follows. In section 5.2 we give an overview of the algorithm and describe some parameters that need to be set. In section 5.3 we describe how we characterize a cell by intersecting it with random lines. In section 5.4 we describe several ways in which our algorithm can be parallelized for implementation on a GPU (Graphics Processing Unit). In section 5.5 we describe how we use machine learning to classify the cells. We present timing and accuracy results in section 5.6. Finally, we provide suggestions for future work in section 5.7 and give our conclusions in section 5.8.

5.2 Technical Approach

We have cultured hBMSCs (human Bone Marrow Stromal Cells) on two types of substrates: a spun-coat substrate with a flat, 2D surface, and a nanofiber substrate with a fibrous, 3D surface. hBMSCs cultured on the nanofiber substrate underwent osteogenic differentiation and adopted a more spikey, branched appearance, while hBMSCs cultured on the spun-coat substrate did not differentiate and retained a smoother appearance. Our goal is to measure three-dimensional geometric properties of the hBMSCs that can distinguish between hBMSCs cultured on the nanofiber substrate and hBMSCs cultured on the spun-coat substrate. Many methods for classifying three-dimensional objects can be found in the literature. For a survey of these methods, see Cardone *et al.* [101].

Our general work-flow is as follows. First, confocal microscopy was used to collect 3D image data sets of individual cells. Next we use an algorithm to intersect each cell with many lines in 3D and record the lengths of each of the partial line segments that run through the interior of the cell. From this line length data, we generate a line length histogram describing each cell. Finally, we use the histograms to train a machine learning technique which is used to classify new cells as coming from a 3D, fibrous nanofiber scaffold or a 2D, flat spun-coat film. An overview of this pipeline is given in figure 5.2.

The 3D cell images that were analyzed in the current work were collected previously and this previous work has been described in detail [86]. Briefly, primary human bone marrow stromal cells (hBMSCs) isolated from iliac crest of healthy

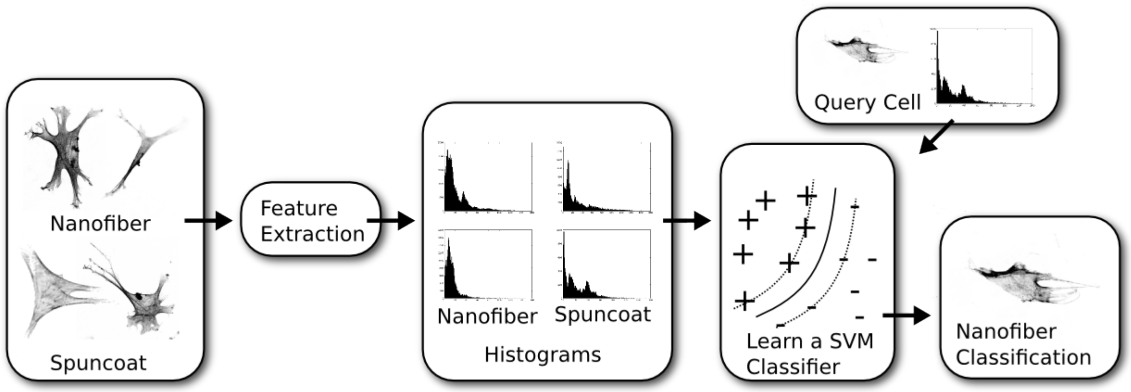


Figure 5.2: Overview of our cell data processing pipeline.

donors were cultured for 1 day on PCL (poly(ε-caprolactone)) spun-coat films or PCL nanofiber scaffolds. 2D PCL spun-coat films were made by spin-coating. 3D PCL nanofibers were made by electrospinning. Nanofiber diameter was measured by imaging with scanning electron microscopy (mean nanofiber diameter = 910 nm, S.D. = 526 nm, n = 100). 2D spun-coat films and 3D nanofiber scaffolds were made from the same material, PCL, so that the effects of scaffold chemistry could be discerned from effects of scaffold structure. For 3D imaging, cellular actin was stained with Alexa-Fluor-546-phalloidin and imaged by confocal fluorescence microscopy. Z-stacks of images of individual hBMSCs on spun-coat films for nanofiber scaffolds were collected using a 1 μm step size.

Previous analysis of hBMSC shape demonstrated that hBMSCs on 3D nanofiber scaffolds have a more branched structure than hBMSCs on the 2D spun-coat films [86]. We want to leverage this difference to distinguish between hBMSCs that were cultured on the different substrates. To accomplish this, we first intersect the cells with a set of randomly generated lines. We measure the lengths of the portions of the lines that passed through the cell and put these lengths into histograms. Our hypothesis was that the cells on the nanofiber substrates would have more short segments since these cells had more long, thin branches. Analysis of the average segment length for the two cell types supports this hypothesis.

A question that arises is how to choose the random lines that are used to intersect with the cells. We want a method of line selection whose results do not depend on microscope orientation. We have tried two different methods of line selection. The first is to simply select pairs of points on a bounding sphere around the cell and generate lines that pass through both of the points. The results of this method should converge to the same result regardless of the cell's orientation. The second method is to select pairs of points on the curved surface of a cylinder rather than the surface of a sphere. This method takes advantage of the particular structure of our data set.

To perform our algorithm we need to classify each voxel in the three-dimensional data set as either being inside or outside the cell. Each voxel contains an integer that represents how much light was received from that location by the microscope. The fluorescently stained cell gives off a large amount of light from locations that are inside the cell, while only some small amount of noise is received from locations outside the cell. We classify the voxels by selecting a threshold value and labeling voxels with values below this threshold as being outside the cell, and voxels with values above this threshold as being inside. If the threshold value is chosen too low then noise in the data set may be labeled as part of the cell, while if the threshold value is chosen too high then significant geometric features of the cell may be eroded.

We therefore expect that, as the threshold value is increased, the performance of the algorithm will improve up to a certain point and then begin to worsen. The selection of the threshold value can be considered part of the training phase of the algorithm and the value can be chosen based on the training data set. For this work the threshold value was chosen by testing a series of different values and selecting the one that produces the greatest percentage of correct classification. Based on the data presented in table 5.1, we selected a threshold value of 3.0.

Threshold	0.5	1.0	2.0	3.0	4.0	5.0	6.0
% Correct	74.2	75.6	80.5	82.9	73.2	78.0	56.1

Table 5.1: Percent correct classification with various threshold values. The voxels in the data set contained values in the range [0, 15]. Parameters used were 10^6 lines intersected, minimum gap length of 8.0, linear SVM, ten-fold cross validation.

In addition to choosing a threshold value we clean the data set by identifying connected components and throwing out all components except the largest. This helps remove noise from the data set and also removes portions of other cells that might have been captured in the images. We fill in any holes (empty regions completely surrounded by cell voxels) we find in the component.

After these steps there can still be some noise in the data set that causes voxels that should be inside the cell to be labeled as empty space. In our ray shooting algorithm we correct this by ignoring gaps in the line segments that are smaller than a certain length, instead counting it as one continuous segment. If this minimum gap length is set too low then it will allow gaps that are actually noise or artifacts, while if it is set too high then it will exclude gaps that are actually part of the cell geometry. We therefore expect that, as the minimum gap length is increased, the performance of the algorithm will improve up to a certain point and then begin to worsen. As with the previously mentioned threshold value, the selection of the minimum gap length can be considered part of the training phase of the algorithm and can be performed based on the training data set. For this work we chose our value for the minimum gap length by once again testing a series of values and selecting the one that produced the greatest percentage of correct classification. Based on the data presented in table 5.2, we selected a minimum gap length of 8.0.

Min Gap Length (μm)	5.0	6.0	7.0	8.0	9.0	10.0	11.0
% Correct	78.0	80.5	80.5	82.9	80.5	80.5	80.5

Table 5.2: Percent correct classification with various minimum gap lengths. Parameters used were 10^6 lines intersected, threshold value of 3.0, linear SVM, ten-fold cross validation.

5.3 Geometric Characterization of the Cell

As described above, our analysis of the cells involves generating a set of random lines that intersect the cell. The lines were generated using several different methods. In the first method we picked pairs of points on the surface of a bounding sphere and generating a line that intersects these points. This is the method used by Juba and Varshney [3] and is described by Li *et al.*[62] as the Chord Model. Uniformly distributed points (x, y, z) on a sphere can be generated from pairs (u, θ) of uniformly distributed random numbers by using the formula

$$(x, y, z) = ((1-u^2)^{\frac{1}{2}}\cos\theta,\ (1-u^2)^{\frac{1}{2}}\sin\theta,\ u) \tag{5.1}$$

where u is in $[-1, 1]$ and θ is in $[0, 2\pi)$. A slightly more computationally efficient formula is given by Rovira *et al.* [66] which generates the points (x, y, z) from pairs (ξ_1, ξ_2) of uniformly distributed random numbers

$$\begin{aligned}
&\cos\theta = 1 - 2 * \xi_1 \\
&\sin\theta = \sqrt{1 - (\cos\theta)^2} \\
&\varphi = 2 * \pi * \xi_2 \\
&(x, y, z) = (\sin\theta * \sin\varphi, \cos\theta, \sin\theta * \cos\varphi)
\end{aligned} \tag{5.2}$$

where ξ_1 and ξ_2 are in $[0, 1)$.

In the second method we picked pairs of points on the curved surface of a cylinder and generate a line that intersects these points. The cylinder is oriented such that the central axis is perpendicular to the plane of the 2D microscope images that were stacked together to form the 3D volume. Each of the two points that define the line is defined by an angle θ around the circumference of the cylinder and a height z along the central axis. This is illustrated in figure 5.3. The formula for points (x, y, z) on the curved surface of a cylinder is

$$(x, y, z) = (\cos\theta, \sin\theta, z) \tag{5.3}$$

where z is in $[-1, 1]$ and θ is in $[0, 2\pi)$.

Both of these line generation methods require a set of uniformly distributed random numbers. Typically a pseudo-random sequence of random numbers is used

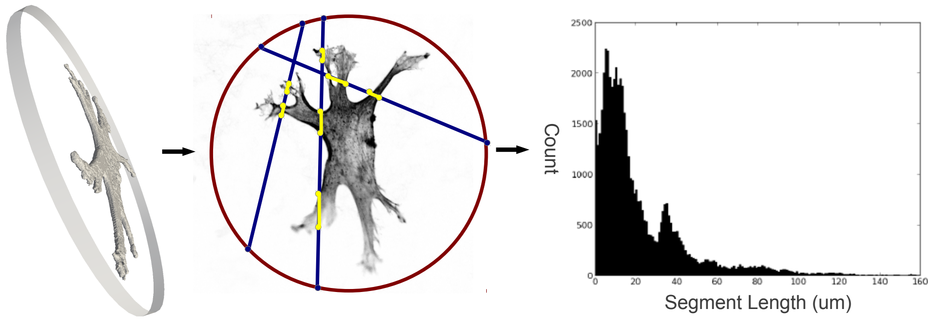


Figure 5.3: Top-down view of lines defined by points on a cylinder enclosing a cell. The lengths of the intersections of these lines with the cell are stored in a histogram.

for this purpose. However, it has been shown that a so-called quasi-random sequence (also called a low-discrepancy sequence) has better properties, including a lower error bound in numeric integration [62]. For this work we use the Niederreiter quasi-random sequence [1], which can be found in the GNU Scientific Library [65]. For equation 5.1, we generate quasi-random points (a, b, c, d) in four dimensions and use the first and second coordinate pairs (a, b) and (c, d) to generate the (u_1, θ_1) and (u_2, θ_2). For equation 5.3, we generate quasi-random points (a, b, c) in three dimensions and use the three coordinates for θ_1, θ_2, and z (in our implementation both points of the line are at the same altitude z).

The sphere method of generating lines is good for general data sets in which no direction should be treated differently than any other. In the data we are working with, however, most of the cell structure variation is in the xy plane of the microscope images, with only minor changes along the z axis along which the images were stacked. We therefore chose to generate lines using the cylinder method with the central axis of the cylinder aligned with the z axis of the image stack when computing the results reported in this chapter. We have also found experimentally that we get better cell classification results with the cylinder method (up to 82.9% correct) than with the sphere method (up to 75.6% correct).

Once a line is generated the next step is to compute its intersections with the cell. We do this by stepping along the line at uniform intervals and at each point checking if that point is inside or outside the cell. Initially, the line starts outside the cell. If two adjacent points are inside and outside the cell respectively (or vice versa), we know that we have either entered or left the cell. Whenever we leave the cell we compute the length of the line segment that was inside the cell and store this in a histogram.

5.4 High-Throughput Processing

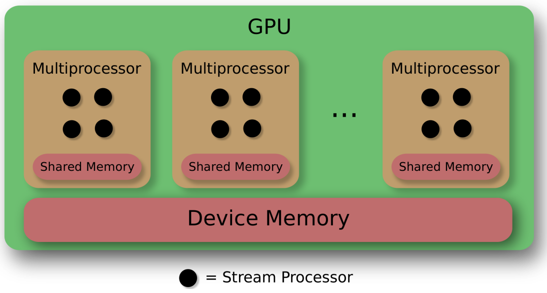


Figure 5.4: GPU memory and multiprocessor layout.

A GPU (Graphics Processing Unit) is a SIMT (Single Instruction, Multiple Thread) processor capable of executing many identical instruction threads in parallel on different sets of input data. It has a large number of stream processors, each with some local memory and registers. These are grouped into several multiprocessors, each containing a small amount of fast memory that is shared within the multiprocessor. It also has a large amount of slower memory that is accessible by all thread processors. A diagram of the processors and memory is given in figure 5.4.

The SIMT architecture is similar to the traditional SIMD (Single Instruction, Multiple Data) except that in SIMT, threads can take diverging branches. In the NVIDIA CUDA library [51] which we used to implement our algorithm, threads are divided into groups of 32 called warps. Threads within a warp cannot execute different instructions at the same time. If threads in a warp go down diverging branches, the GPU will first disable all the threads going down the second branch and execute the instructions of the first branch, then vice versa. Greater efficiency can therefore be achieved by ensuring that as often as possible all threads in a warp follow the same branches. Threads in different warps are free to go down different branches without penalty.

If an algorithm can be mapped to this architecture, the GPU can generally execute the algorithm much faster than it could be executed on a CPU. This is because the SIMT nature of the GPU allows more of its transistors to be dedicated to arithmetic operations rather than tasks such as caching and flow control.

We mapped our line intersection algorithm to the GPU by using one thread to compute the intersections of each line with the cell. A straightforward implementation of this algorithm involves each thread writing the lengths of the intersecting segments into a shared histogram (see figure 5.5). However, this requires synchronization of the threads or the use of slow atomic operations. One way around this

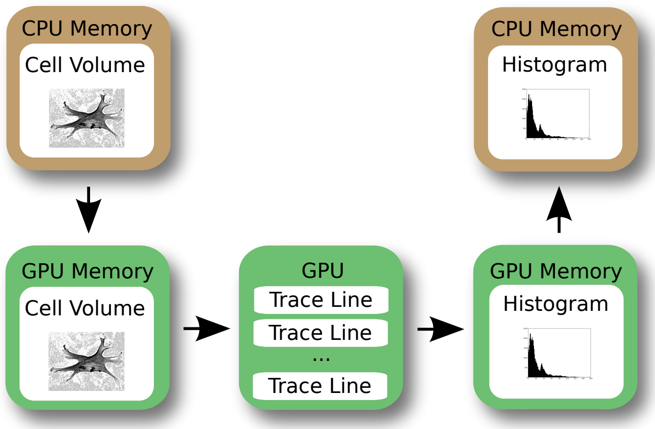


Figure 5.5: Use of CPU and GPU to generate segment length histogram from cell volume using atomic operations algorithm.

requirement is to give each thread its own 256 bin histogram to store its results to, and then merge the histograms at the end using a parallel reduction operation (see figure 5.6). This approach is wasteful, however, since each thread would likely write only a few segment lengths into each histogram. Instead, each thread writes its intersected segment lengths into a fixed size list, and simply ignores any intersected segments that occur after the list is full. We found that a list of length 10 is sufficient to produce results that are almost identical to the CPU version of the algorithm. Once the lists are constructed they are read back to the CPU and the lengths are counted to generate the histogram (see figure 5.7). A comparison of performance data for these three algorithms is given in section 5.6.

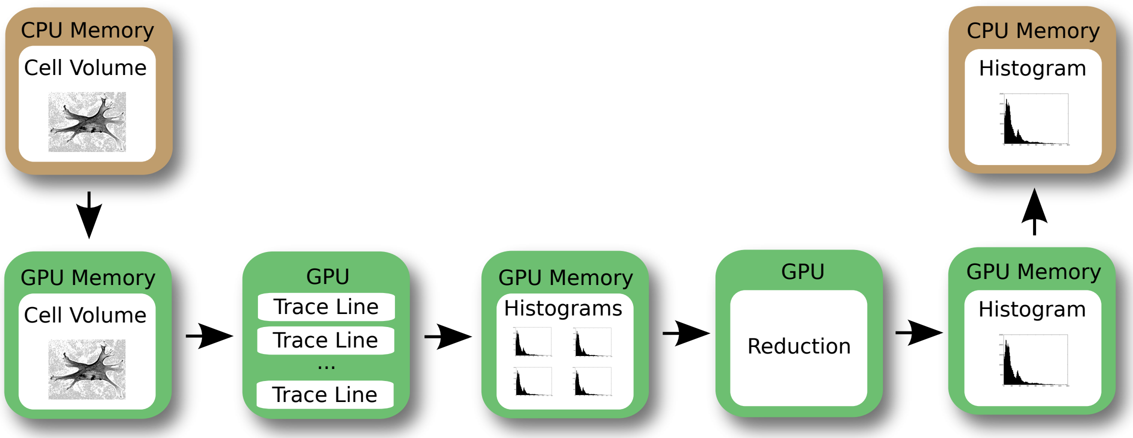


Figure 5.6: Use of CPU and GPU to generate segment length histogram from cell volume using parallel reduction algorithm.

5.5 Classification Through Machine Learning

The segment length measurements produced histograms for the nanofiber and spun-coat cells. To show that these measurements could distinguish between the two types of cells we applied several machine learning algorithms from the Weka machine learning toolkit [102]. The algorithms we used included Naive Bayes, Support Vector Machine, k Nearest Neighbor, and Decision Tree. For each algorithm we used ten-fold cross validation. This means we divided the data into ten groups, used nine groups to train the algorithm, and then classified the tenth. This was repeated using each of the ten groups as the classification group. The results of the ten classifications were then combined to give classification results for the whole data set. We obtained the best classification results using Support Vector Machines (SVM). Once the decision was made to use SVM, we switched from using Weka to using LIBSVM [103].

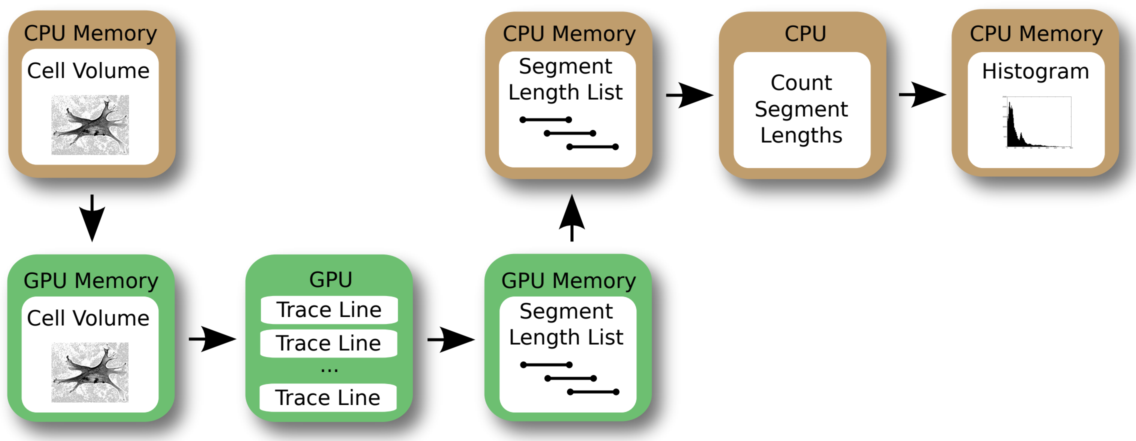


Figure 5.7: Use of CPU and GPU to generate segment length histogram from cell volume using segment length list algorithm.

The training phase of the SVM treats each histogram as a point in high-dimensional space and tries to compute the hyperplane that best separates the points corresponding to differentiated cells and the points corresponding to non-differentiated cells. The orientation of the hyperplane is determined only by those points close to it, which are referred to as the support vector. Once the algorithm is trained, additional histograms can be classified as differentiated or non-differentiated by testing where the corresponding point lies in relation to the hyperplane. An illustration of a trained SVM is given in figure 5.8. Additional information on Support Vector Machines can be found by referring to Cortes and Vapnik [104].

5.6 Performance and Validation

We tested our algorithm on a data set developed by Kumar *et al.* [86] consisting of 21 cells grown on a fibrous nanofiber substrate and 20 cells grown on a flat

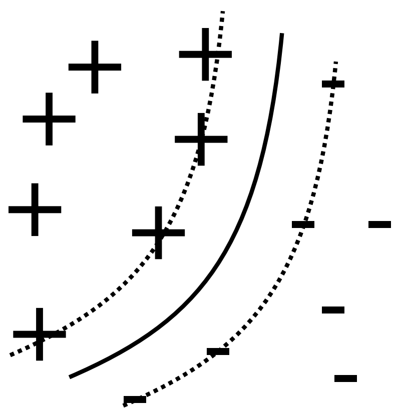


Figure 5.8: Trained Support Vector Machine (SVM). The hyperplane that separates the two classes of histograms is the zero level-set of a decision function that divides space into a positive half and a negative half. Histograms that fall in the positive half are labeled as one class while histograms that fall in the negative half are labeled as the other.

spun-coat substrate. For each cell we were given a stack of 16-bit grayscale confocal microscopy images of resolution 512×512 which we merged together into a single volume. The number of images in each stack varied from 11 to 20.

We measured the performance of our algorithms when intersecting a cell with various numbers of lines, ranging from 10^3 to 10^6. The results are given in table 5.3. For each algorithm the first step was to generate the random lines with which to intersect the cell. This step is independent of the data and can be done once as a pre-process, with the same set of lines then being used to intersect each cell. The line generation time is therefore not included in the running times for the algorithms.

For each number of lines we measured the times the algorithms took to compute the intersection counts for a single cell both on the CPU and in the parallel implementations on the GPU. The timing for each algorithm is broken down into the steps listed in figures 5.5, 5.6, and 5.7. For the GPU Reduction algorithm we measured the time required to send the lines to the GPU and trace them ("Trace lines"), as well as the time required to perform the parallel reduction operation and read back the resulting histogram ("Do reduction"), which was implemented using the CUDA Thrust library [105]. For the GPU Lists algorithm we again measured the time required to send the lines to the GPU and trace them ("Trace lines"), as well as the time required to read the segment length lists back to the CPU and convert them into a histogram ("Count lengths"). The total time includes these times as well as the time required for any other miscellaneous tasks. In addition to these times, all algorithms took about 270 ms to load the cell volume data from the disk. We also list speedup factors which show how many times faster each parallel GPU implementation is over the serial implementation on the CPU. Note that the best parallel GPU implementation can be over two orders of magnitude faster than the serial CPU implementation.

On the current-generation GPU listed in the caption of table 5.3 the best performing algorithm was Atomic, followed closely by Lists. In addition to this GPU we also tested the algorithms on several older-generation GPUs, an NVIDIA Quadro NVS 285 and an NVIDIA Quadro NVS 290. On these GPUs the Lists algorithm was actually slightly faster than Atomic. We suspect the reason for this is that the implementation of atomic operations has been improved in the current generation of GPUs.

Once the cell histograms have been produced they are either used to train an

Num lines	10^3	10^4	10^5	10^6
Generate lines (ms)	0.163	1.35	12.9	126
CPU Algorithm				
Total time (ms)	50.1	492	4915	49154
GPU Algorithm – Atomic				
Total time (ms)	3.56	6.74	45.7	450
Speedup factor	14.1	73.0	108	109
GPU Algorithm – Reduction				
Trace lines (ms)	0.388	0.467	1.55	17.1
Do reduction (ms)	13.4	19.1	80.0	708
Total time (ms)	14.1	20.1	83.4	743
Speedup factor	3.55	24.5	58.9	66.2
GPU Algorithm – Lists				
Trace lines (ms)	0.277	0.450	1.55	15.9
Count lengths (ms)	2.90	6.48	49.2	466
Total time (ms)	3.52	7.35	52.7	501
Speedup factor	14.2	66.9	93.3	98.1

Table 5.3: Results of intersecting a cell with different numbers of lines. The CPU algorithm was run on an Intel Xeon X5260 (using only one core) with 8 GB of RAM. The GPU algorithms were run on an NVIDIA Tesla C2050. The dimensions of the cell volume data were $512 \times 512 \times 20$. The threshold was 3.0 and the minimum gap length was 8.0.

SVM classifier (if they are in the training set), or are classified by a trained SVM. For this work we used the SVM implementation in LIBSVM [103]. Training a linear SVM on our test data set of 41 cell histograms took about 23.9 ms. Once the SVM is trained, new cell histograms can be classified in about 17.4 ms each.

In addition to running time we also measured the classification correctness of our algorithm when intersecting the cells with different numbers of lines. To compute the percentage of correct classification for each trial we used 10-fold cross validation. This means that we divided the data set of 41 cells into 10 groups of approximately equal size, trained the machine learning algorithm on 9 of the groups, and measured the percent correct classification of the 10th. An example of one fold is given in table 5.4. This was repeated 10 times, each time using a different group to test the classification correctness. The percent correct classification of all the groups was then averaged. The average percent correct classifications are given in figure 5.9.

This data demonstrates the progressive nature of our algorithm. For a small number of lines the classification accuracy is about 50%, which is what would be expected from random guessing. As the number of intersected lines increases, so does the classification accuracy until it levels off at a maximum. If, after a certain number of lines have been intersected, the user desires additional accuracy, then the already-computed lines can be re-used and only the additional lines will need to be intersected. The algorithm can therefore provide rough results quickly, which can then be improved to the desired accuracy with additional running time.

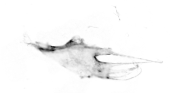
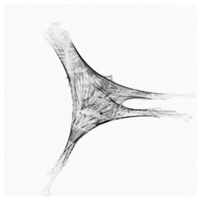
				
Actual	Nanofiber	Nanofiber	Spun-Coat	Spun-Coat
Classified as	Nanofiber	Spun Coat	Spun-Coat	Spun-Coat

Table 5.4: Example of one fold of the validation of a trained SVM model. Histograms were generated using the GPU Lists algorithm with threshold 3.0 and minimum gap length 8.0. An SVM was trained with 19 nanofiber cells and 18 spun-coat cells. An additional 2 nanofiber cells and 2 spun-coat cells were set aside to be classified by the model. Of these 4 cells, 3 were classified correctly.

5.7 Future Work

One interesting area of potential future work would be to take advantage of the progressive nature of the algorithm during the SVM classification. Rather than simply generating the histogram using the full number of lines and then performing the classification, we could instead generate the histogram using some small initial number of lines and then check the certainty of the classification. If the classification was still doubtful then the histogram could be improved by intersecting the cell with more lines, while if the classification was sufficiently certain then the results could be returned immediately. For an SVM, the certainty of the classification could perhaps be measured by the distance of the query point from the hyperplane dividing the two regions of classification.

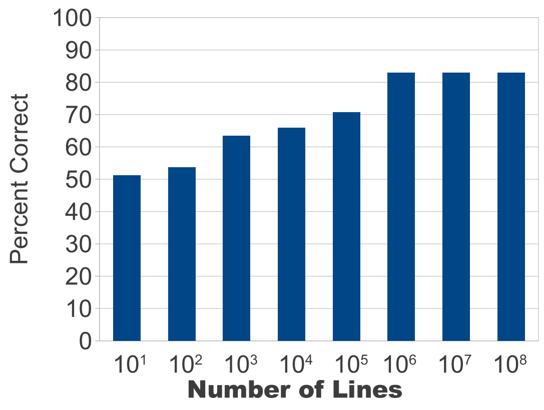


Figure 5.9: Percent correct classification of differentiated and non-differentiated stem cells. This data was generated using the GPU Lists algorithm with threshold 3.0 and minimum gap length 8.0.

Although we have only applied the algorithm to cell data, the algorithm is general enough that it could potentially be applied to any type of 3D data, such as CAD models or protein molecules. The use of the algorithm to classify other data types could be another interesting avenue for future work.

5.8 Conclusions

We described an algorithm that can classify hBMSCs (human Bone Marrow Stromal Cells) as having been grown on a differentiation-inducing 3D PCL nanofiber scaffold or on a non-differentiation-inducing 2D PCL spun-coat film. The algorithm takes 3D cell image data and intersects it with randomly generated rays that connect the sides of a cylinder that bounds the cell. The lengths of ray segments that are within the cell are used to generate a histogram. These histograms can then be used as sample points to train a machine learning algorithm such as a support vector machine (SVM), which can then be used to classify future cells.

Our algorithm is easily parallelizable and is also progressive, allowing it to provide a rough histogram quickly and then refine it as desired. Our parallel GPU implementation can convert a cell into a histogram representation suitable for machine learning training or classification by intersecting it with 10^6 lines in about 450 ms, representing an over 100-fold speedup from the serial CPU implementation. By applying the algorithm to our test data set of 41 cells we were able to achieve 82.9% correct classification using 10-fold cross validation. This rapid 3D image analysis algorithm can be used to classify differentiating and non-differentiating stem cells for high-throughput screening of 3D tissue scaffolds. The algorithm used 3D cell image data in order to take advantage of the benefits of 3D culture and to capture the effects of 3D scaffold structure on cell shape. The approach has been demonstrated using stem cell image data from 1-day cultures, which enables identification of differentiating cells at a much earlier stage than is possible with osteogenic markers, which can require weeks of culture

Chapter 6

Future Work

A current trend in science is the generation of large amounts of data, both from measurement and from computer simulation. Various sources have referred to this phenomenon as the generation of "Big Data". At the same time, computer processor clock speeds have stopped increasing, forcing hardware manufacturers to resort to parallelism to continue to increase processing power. Unfortunately, algorithms designed for the traditional serial processing model are often not able to take advantage of this parallel computing ability. To keep up with Big Data, we must design future algorithms using data-parallel techniques.

There are several potential next steps that could be taken from the work described in this thesis. One possibility would be to combine the Gaussian Radial Basis Function (RBF) fitting from Chapter 2 with the stem cell classification from Chapter 5. The use of RBFs could enable several different improvements in the classification algorithm.

The first improvement would involve the use of an analytical Gaussian blur of the RBF representation. As different levels of blur are applied, the cell shape would change in a certain way that could be characteristic of a differentiated or non-differentiated cell. The line intersection length histograms from Chapter 5 could then be generated for each of these different blur levels, and used as additional information for the Support Vector Machine (SVM) training and classification.

The second improvement would involve the use of the Gaussian RBF's analytic differentiability to compute the mean curvature at random points on the cell's implicit surface. Mean curvatures of the implicit surface could be computed directly from the volumetric data using the formulas from Goldman [106]. These curvature samples could then be put into histograms just as the line intersection lengths were in Chapter 5, and used as additional information for the classification of the cells.

Another potential piece of future work would be to combine the Confluent Visualization from Chapter 4 with the stem cell and surface data from Chapter 5. Biologists believe that the geometry of the material which a stem cell rests on can induce the cell to undergo differentiation. It would therefore be useful to study the geometric interaction between the cell surface and the surface of the material on which it is resting. Confluent Visualization could enable this by allowing only the portions of one surface that are in contact with the other surface to be visualized.

A final direction for potential future work is in the area of cybersecurity. Companies, governments, and consumers depend on secure and reliable computer networks and data products, but as technology becomes more complex, security threats also become more complicated. The scale of network traffic data is truly staggering and our ability to collect such data has far surpassed our ability to meaningfully analyze it. Visual representations and interaction technologies provide a powerful mechanism for allowing an analyst to see and understand large amounts of information at once. The development of data-parallel visual representation algorithms could be an important step towards the solution of this problem.

Bibliography

[1] Paul Bratley, Bennett L. Fox, and Harald Niederreiter. Implementation and tests of low-discrepancy sequences. *ACM Trans. Model. Comput. Simul.*, 2(3):195–213, 1992.

[2] D. Juba and A. Varshney. Modelling and rendering large volume data with gaussian radial basis functions. Technical report, Tech. rep., University of Maryland, 2007.

[3] D. Juba and A. Varshney. Parallel, stochastic measurement of molecular surface area. *Journal of Molecular Graphics and Modelling*, 27(1):82–87, 2008.

[4] G. Stantchev, D. Juba, W. Dorland, and A. Varshney. Confluent volumetric visualization of gyrokinetic turbulence. *Plasma Science, IEEE Transactions on*, 36(4):1112–1113, 2008.

[5] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1999. ACM Press.

[6] James F. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[7] Hitoshi Nishimura, Makoto Hirai, Toshiyuki Kawai, Toru Kawata, Isao Shirakawa, and Koichi Omura. Object modeling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, J68-D(4):718–725, 1985. In Japanese (translated into English by Takao Fujiwara while at Centre for Advanced Studies in Computer Aided Art and Design, Middlesex Polytechnic, England, 1989).

[8] J. Bloomenthal, C. Bajaj, J. Blinn, M. Cani-Gascuel, A. Rockwood, B. Wyvill, and G. Wywill. *Introduction to Implicit Surfaces.* Morgan Kaufmann, 1997.

[9] Manfred Weiler, Ralf Botchen, Simon Stegmaier, Thomas Ertl, Jingshu Huang, Yun Jang, David S. Ebert, and Kelly P. Gaither. Hardware-assisted feature analysis and visualization of procedurally encoded multifield volumetric data. *IEEE Comput. Graph. Appl.*, 25(5):72–81, 2005.

[10] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 255–262, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[12] J. C. Carr, W. R. Fright, and R. K. Beatson. Surface interpolation with radial basis functions for medical imaging. *IEEE Transactions on Medical Imaging*, 16(1):96–107, February 1997.

[13] E. W. Cheney and W. A. Light. *A Course in Approximation Theory.* Brooks Cole, Pacific Grove, 1999.

[14] N. Dyn, D. Levin, and S. Rippa. Numerical procedures for surface fitting of scattered data by radial basis functions. *SIAM Journal on Scientific and Statistical Computing*, 7(2):639–659, 1986.

[15] Bryan S. Morse, Terry S. Yoo, Penny Rheingans, David T. Chen, and K. R. Subramanian. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Shape Modeling and Applications, SMI 2001 International Conference on*, pages 89–98, 2001.

[16] G. Turk, Huong Quynh Dinh, J. F. O'Brien, and G. Yngve. Implicit surfaces that interpolate. In Bob Werner, editor, *Proceedings of the International Conference on Shape Modeling and Applications (SMI-01)*, pages 62–73, Los Alamitos, CA, May 7–11 2001. IEEE Computer Society.

[17] Greg Turk and James F. O'Brien. Shape transformation using variational implicit functions. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[18] Greg Turk and James F. O'Brien. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4):855–873, 2002.

[19] G. Yngve and G. Turk. Creating smooth implicit surfaces from polygonal meshes. *Technical Report GIT-GVU-99-42, Graphics, Visualization, and Usability Center. Georgia Institute of Technology, 1999.*, 1999.

[20] J. Duchon. Splines minimizing rotation-invariant semi-norms in sobolev spaces. *In W. Schempp and K. Zeller, editors, Constructive Theory of Functions of Several Variables, number 571 in Lecture Notes in Mathematics*, pages 85–100, 1977.

[21] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 67–76, New York, NY, USA, 2001. ACM Press.

[22] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[23] Christopher S. Co, Bjoern Heckel, Hans Hagen, Bernd Hamann, and Kenneth I. Joy. Hierarchical clustering for unstructured volumetric scalar fields. In Greg Turk, Jarke J. van Wijk, and Robert Moorhead, editors, *Proceedings of IEEE Visualization 2003*, pages 325–332. IEEE, October 19–24 2003.

[24] Yun Jang, Manfred Weiler, Matthias Hopf, Jingshu Huang, David S. Ebert, Kelly P. Gaither, and Thomas Ertl. Interactively visualizing procedurally encoded scalar fields. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2004)*, 2004.

[25] I. T. Jolliffe. *Principal Component Analysis.* Springer-Verlag, 1986.

[26] W. Hong, N. Neophytou, K. Mueller, and A. Kaufman. Constructing 3d elliptical gaussians for irregular data. In *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, 2006.

[27] Yun Jang, Ralf P. Botchen, Andreas Lauser, David S. Ebert, Kelly P. Gaither, and Thomas Ertl. Enhancing the interactive visualization of procedurally encoded multifield data with ellipsoidal basis functions. *Eurographics*, 25(3), 2006.

[28] John C. Hart. Ray tracing implicit surfaces. In *SIGGRAPH 93 Modeling, Visualizing, and Animating Implicit Surfaces course notes*, pages 13–1 to 13–15. ACM SIGGRAPH, August 1993.

[29] Devendra Karla and Alan H. Barr. Guaranteed ray intersections with implicit surfaces. *Computer Graphics*, 23(3):297–306, 1989.

[30] Andrei Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2), June 1999.

[31] Geoff Wyvill and Andrew Trotman. Ray-tracing soft objects. In *New Trends in Computer Graphics (Proceedings of CG International '90)*, pages 467–476. Springer-Verlag, 1990.

[32] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277, New York, NY, USA, 1994. ACM Press.

[33] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[34] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, November 1988.

[35] Jules Bloomenthal. An implicit surface polygonizer. In Paul Heckbert, editor, *Graphics Gems IV*, pages 324–349. Academic Press, Boston, 1994.

[36] N. Neophytou, K. Mueller, K. McDonnell, W. Hong, X. Guan, H. Qin, and A. Kaufman. Gpu-accelerated volume splatting with elliptical rbfs. In *Joint Eurographics - IEEE TCVG Symposium on Visualization 2006 (EuroVis'06)*, 2006.

[37] Steven M. Kay. *Fundamentals of Statistical Signal Processing: Estimation Theory*, chapter 7. Prentice Hall, 1993.

[38] National Library of Medicine (U.S.) Board of Regents. Electronic imaging: Report of the board of regents. U.S. Department of Health and Human Services, Public Health Service, National Institutes of Health, 1990. NIH Publication 90-2197.

[39] Sarah F. Frisken and Ron Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.

[40] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Trans. Graph.*, 22(3):463–470, 2003.

[41] O. Kreylos, N. Max, B. Hamann, S. Crivelli, and W. Bethel. Interactive protein manipulation. In *Proceedings of IEEE Visualization*, pages 581–588, October 2003.

[42] O.D. Lampe, I. Viola, N. Reuter, and H. Hauser. Two-Level Approach to Efficient Visualization of Protein Dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1616–1623, 2007.

[43] M.L. Connolly. Analytical molecular surface calculation. *Journal of Applied Crystallography*, 16(5):548–558, 1983.

[44] ML Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709, 1983.

[45] Michel F. Sanner, Arthur J. Olson, and Jean-Claude Spehner. Fast and robust computation of molecular surfaces. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 406–407, New York, NY, USA, 1995. ACM Press. http://www.scripps.edu/∼sanner/html/msms_home.html.

[46] Robert Fraczkiewicz and Werner Braun. Exact and efficient analytical calculation of the accessible surface areas and their gradients for macromolecules. *Journal of Computational Chemistry*, 19(3):319–333, 1998. http://pauli.utmb.edu/cgi-bin/get_a_form.tcl.

[47] Tolga Can, Chao-I Chen, and Yuan-Fang Wang. Efficient molecular surface generation using level-set methods. *J Mol Graph Model*, 25(4):442–454, December 2006.

[48] N. R. Voss, M. Gerstein, T. A. Steitz, and P. B. Moore. The geometry of the ribosomal polypeptide exit tunnel. *J Mol Bio*, 360(4):893–906, July 2006. http://geometry.molmovdb.org/3v/.

[49] Chandrajit Bajaj and Vinay Siddavanahalli. An adaptive grid based method for computing molecular surfaces and properties. Technical Report tr06-56, University of Texas, 2006.

[50] Chandrajit Bajaj, Peter Djeu, Vinay Siddavanahalli, and Anthony Thane. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 243–250, Washington, DC, USA, 2004. IEEE Computer Society.

[51] http://developer.nvidia.com/object/cuda.html.

[52] http://ati.amd.com/companyinfo/researcher/Documents.html.

[53] B. Lee and FM Richards. The interpretation of protein structures: estimation of static accessibility. *J Mol Biol*, 55(3):379–400, 1971.

[54] F.M. Richards. Areas, Volumes, Packing, and Protein Structure. *Annual Review of Biophysics and Bioengineering*, 6(1):151–176, 1977.

[55] H. Edelbrunner and EP Muecke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43–72, 1994.

[56] Shoshana J. Wodak and Joel Janin. Analytical approximation to the accessible surface area of proteins. *Proc Natl Acad Sci USA*, 77(4):1736–1740, April 1980.

[57] A. Varshney, F. P. Brooks, and W. V. Wright. Linearly scalable computation of smooth molecular surfaces. *IEEE Computer Graphics and Applications*, 14, No. 5:19 – 25, 1994.

[58] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1(3):235–256, 1982.

[59] J. A. Grant and B. T. Pickup. A gaussian description of molecular shape. *Journal of Physical Chemistry*, 99(11):3503–3510, December 1995.

[60] David W. Ritchie. Evaluation of protein docking predictions using hex 3.1 in capri rounds 1 and 2. *Proteins: Structure, Function, and Genetics*, 52(1):98–106, July 2003.

[61] H.M.Berman, J.Westbrook, Z.Feng, G.Gilliland, T.N.Bhat, H.Weissig, I.N.Shindyalov, and P.E.Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000. http://www.pdb.org/.

[62] Xueqing Li, Wenping Wang, and Adrian Bowyer. Using low-discrepency sequences and the crofton formula to compute surface areas of geometric models. *Computer-Aided Design*, 35(9):771–82, August 2003.

[63] Yu-Shen Liu, Jun-Hai Yong, Hui Zhang, Dong-Ming Yan, and Jia-Guang Sun. A quasi-monte carlo method for computing areas of point-sampled surfaces. *Computer-Aided Design*, 38(1):55–68, January 2006.

[64] Eric W. Weisstein. Sphere point picking. From MathWorld– A Wolfram Web Resource, http://mathworld.wolfram.com/SpherePointPicking.html.

[65] Gnu scientific library. http://www.gnu.org/software/gsl/.

[66] J. Rovira, P. Wonka, F. Castro, and M. Sbert. Point sampling with uniformly distributed lines. In *Point-Based Graphics, 2005. Eurographics/IEEE VGTC Symposium Proceedings*, pages 109–118, 2005.

[67] Bernd Gärtner. Fast and robust smallest enclosing balls. In *ESA '99: Proceedings of the 7th Annual European Symposium on Algorithms*, pages 325–338, London, UK, 1999. Springer-Verlag.

[68] Markus Gross and Hanspeter Pfister, editors. *Point-Based Graphics*. Morgan Kaufmann, 2007.

[69] Peter Schröder. What can we measure? In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 5–9, New York, NY, USA, 2006. ACM Press.

[70] J. Liang, H. Edelsbrunner, P. Fu, P.V. Sudhakar, and S. Subramaniam. Analytical shape computation of macromolecules: II. Inaccessible cavities in proteins. *Proteins Structure Function and Genetics*, 33(1):18–29, 1998.

[71] A. M. Dimits, G. Bateman, M. A. Beer, B. I. Cohen, W. Dorland, G. W. Hammett, C. Kim, J. E. Kinsey, M. Kotschenreuther, A. H. Kritz, L. L. Lao, J. Mandrekas, W. M. Nevins, S. E. Parker, A. J. Redd, D. E. Shumaker, R. Sydora, and J. Weiland. Comparisons and physics basis of tokamak transport models and turbulence simulations. *Physics of Plasmas*, 7(3):969–983, 2000.

[72] W. Dorland, F. Jenko, M. Kotschenreuther, and B. N. Rogers. Electron temperature gradient turbulence. *Phys. Rev. Lett.*, 85(26):5579–5582, Dec 2000.

[73] M. Kotschenreuther, G. Rewoldt, and W. Tang. *Comput. Phys. Commun*, 88(128), 1995.

[74] Derek Juba, Antonio Cardone, Horace Ip, Carl G. Simon Jr., Chris K. Tison, Girish Kumar, Mary Brady, and Amitabh Varshney. Parallel geometric classification of stem cells by their 3d morphology.

[75] Ana Jaklenec, Andrea Stamp, Elizabeth Deweerd, Angela Sherwin, and Robert Langer. Progress in the tissue engineering and stem cell industry are we there yet?. *Tissue Engineering Part B: Reviews*, 18(3):155–166, 2012.

[76] H.C. Ott, T.S. Matthiesen, S.K. Goh, L.D. Black, S.M. Kren, T.I. Netoff, and D.A. Taylor. Perfusion-decellularized matrix: using nature's platform to engineer a bioartificial heart. *Nature medicine*, 14(2):213–221, 2008.

[77] A. Atala, S.B. Bauer, S. Soker, J.J. Yoo, and A.B. Retik. Tissue-engineered autologous bladders for patients needing cystoplasty. *The lancet*, 367(9518):1241–1246, 2006.

[78] Atlantida Raya-Rivera, Diego R Esquiliano, James J Yoo, Esther Lopez-Bayghen, Shay Soker, and Anthony Atala. Tissue-engineered autologous urethras for patients who need reconstruction: an observational study. *The lancet*, 377(9772):1175–1182, April 2011.

[79] Robert Langer and Joseph P. Vacanti. Tissue engineering. *Science*, 260:920–925, 1993.

[80] D.S.W. Benoit, M.P. Schwartz, A.R. Durney, and K.S. Anseth. Small functional groups for controlled differentiation of hydrogel-encapsulated human mesenchymal stem cells. *Nature materials*, 7(10):816–823, 2008.

[81] J. Kohn, W.J. Welsh, and D. Knight. A new approach to the rationale discovery of polymeric biomaterials. *Biomaterials*, 28(29):4171, 2007.

[82] T.A. Petrie, J.E. Raynor, D.W. Dumbauld, T.T. Lee, S. Jagtap, K.L. Templeman, D.M. Collard, and A.J. García. Multivalent integrin-specific ligands enhance tissue healing and biomaterial integration. *Science translational medicine*, 2(45):45ra60, 2010.

[83] A.J. Engler, S. Sen, H.L. Sweeney, and D.E. Discher. Matrix elasticity directs stem cell lineage specification. *Cell*, 126(4):677–689, 2006.

[84] S.H. Parekh, K. Chatterjee, S. Lin-Gibson, N.M. Moore, M.T. Cicerone, M.F. Young, and C.G. Simon. Modulus-driven differentiation of marrow stromal cells in 3d scaffolds that is independent of myosin-based cytoskeletal tension. *Biomaterials*, 32(9):2256–2264, 2011.

[85] M.J. Dalby, N. Gadegaard, R. Tare, A. Andar, M.O. Riehle, P. Herzyk, C.D.W. Wilkinson, and R.O.C. Oreffo. The control of human mesenchymal cell differentiation using nanoscale symmetry and disorder. *Nature materials*, 6(12):997–1003, 2007.

[86] G. Kumar, C.K. Tison, K. Chatterjee, P.S. Pine, J.H. McDaniel, M.L. Salit, M.F. Young, and C.G. Simon Jr. The determination of stem cell fate by 3d scaffold structures through the control of cell shape. *Biomaterials*, 32(35):9188–9196, 2011.

[87] G. Kumar, M.S. Waters, T.M. Farooque, M.F. Young, and C.G. Simon. Freeform fabricated scaffolds with roughened struts that enhance both stem cell proliferation and differentiation by controlling cell shape. *Biomaterials*, 2012.

[88] M. Dominici, K. Le Blanc, I. Mueller, I. Slaper-Cortenbach, FC Marini, DS Krause, RJ Deans, A. Keating, DJ Prockop, and EM Horwitz. Minimal criteria for defining multipotent mesenchymal stromal cells. the international society for cellular therapy position statement. *Cytotherapy*, 8(4):315–317, 2006.

[89] J. Makower, A. Meer, and L. Denend. Fda impact on us medical technology innovation: a survey of over 200 medical technology companies. *Arlington (Virginia): National Venture Capital Association*, 2010.

[90] C.S. Chen, M. Mrksich, S. Huang, G.M. Whitesides, and D.E. Ingber. Geometric control of cell life and death. *Science*, 276(5317):1425–1428, 1997.

[91] J. Folkman and A. Moscona. Role of cell shape in growth control. 273, 1978.

[92] R. McBeath, D.M. Pirone, C.M. Nelson, K. Bhadriraju, and C.S. Chen. Cell shape, cytoskeletal tension, and rhoa regulate stem cell lineage commitment. *Developmental cell*, 6(4):483–495, 2004.

[93] M.D. Treiser, E.H. Yang, S. Gordonov, D.M. Cohen, I.P. Androulakis, J. Kohn, C.S. Chen, and P.V. Moghe. Cytoskeleton-based forecasting of stem cell lineage fates. *Proceedings of the National Academy of Sciences*, 107(2):610–615, 2010.

[94] J.T. Elliott, M. Halter, A.L. Plant, J.T. Woodward, K.J. Langenbach, and A. Tona. Evaluating the performance of fibrillar collagen films formed at polystyrene surfaces as cell culture substrates. *Biointerphases*, 3(2):19–28, 2008.

[95] J. Lovmand, J. Justesen, M. Foss, R.H. Lauridsen, M. Lovmand, C. Modin, F. Besenbacher, F.S. Pedersen, and M. Duch. The use of combinatorial topographical libraries for the screening of enhanced osteogenic expression and mineralization. *Biomaterials*, 30(11):2015–2022, 2009.

[96] J. Simon, G. Carl, Y. Yang, V. Thomas, S.M. Dorsey, and A.W. Morgan. Cell interactions with biomaterials gradients and arrays. *Combinatorial Chemistry & High Throughput Screening*, 12(6):544–553, 2009.

[97] F.S. Collins. Reengineering translational science: the time is right. *Science translational medicine*, 3(90):90cm17–90cm17, 2011.

[98] J.H. Sung and M.L. Shuler. A micro cell culture analog (μcca) with 3-d hydrogel culture of multiple cell lines to assess metabolism-dependent cytotoxicity of anti-cancer drugs. *Lab Chip*, 9(10):1385–1394, 2009.

[99] E. Krausz, R. de Hoogt, E. Gustin, F. Cornelissen, T. Grand-Perret, L. Janssen, N. Vloemans, D. Wuyts, S. Frans, A. Axel, et al. Translation of a tumor microenvironment mimicking 3d tumor growth co-culture assay platform to high-content screening. *Journal of Biomolecular Screening*, 2012.

[100] M. Vinci, S. Gowan, F. Boxall, L. Patterson, M. Zimmermann, C. Lomas, M. Mendiola, D. Hardisson, S.A. Eccles, et al. Advances in establishment and analysis of three-dimensional tumor spheroid-based functional assays for target validation and drug evaluation. *BMC biology*, 10(1):29, 2012.

[101] A. Cardone, R.K. Gupta, and M. Karnik. A survey of shape similarity assessment algorithms for product design and manufacturing applications. In *Journal of Computing and Information Science in Engineering*. Citeseer, 2003.

[102] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[103] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[104] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[105] Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2010. Version 1.3.0.

[106] R. Goldman. Curvature formulas for implicit curves and surfaces. *Computer Aided Geometric Design*, 22(7):632–658, 2005.