RICE UNIVERSITY

Techniques for Realtime Viewing and Manipulation of Volumetric Data

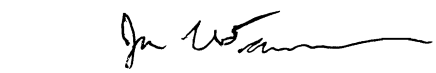
by

Travis McPhail

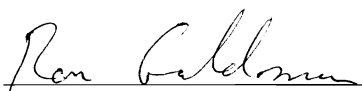
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

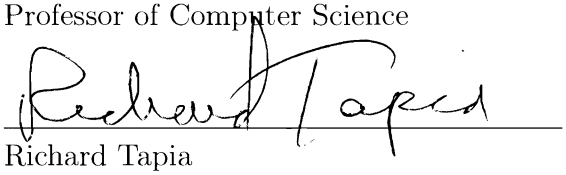
Approved, Thesis Committee:



Joe Warren, Chair
Professor of Computer Science



Ron Goldman
Professor of Computer Science



Richard Tapia
University Professor of Computational
and Applied Mathematics

Houston, Texas

August, 2010

ABSTRACT

Techniques for Realtime Viewing and Manipulation of Volumetric Data

by

Travis McPhail

Visualizing and manipulating volumetric data is a major component in many areas including anatomical registration in biomedical fields, seismic data analysis in the oil industry, machine part design in computer-aided geometric design, character animation in the movie industry, and fluid simulation. These industries have to meet the demands of the times and be able to make meaningful assertions about the data they generate. The shear size of this data presents many challenges to facilitating realtime interaction. In the recent decade, graphics hardware has become increasingly powerful and more sophisticated which has introduced a new realm of possibilities for processing volumetric data.

This thesis focuses on a suite of techniques for viewing and editing volumetric data that efficiently use the processing power of central processing units (CPUs) as well as the large processing power of the graphics hardware (GPUs). This work begins with an algorithm to improve the efficiency of a texture-based volume rendering. We continue with a framework for performing realtime constructive solid geometry (CSG) with complex shapes and smoothing operations on watertight meshes based on a variation of Depth Peeling. We then move to an intuitive technique for deforming volumetric data using a collection of control points. Finally, we apply this technique to image registration of 3-dimensional computed tomography (CT) images used for

lung cancer treatment planning.

Acknowledgements

I wish to express tremendous gratitude to my advisor, Dr. Joe Warren. His immense knowledge and insightful guidance throughout my graduate career has been invaluable. I would also like to thank Dr. Ron Goldman for his suggestions throughout the years. I wish to thank Dr. Devika Subramanian for reading my thesis and participating in my thesis defense. In addition, I would like to thank Dr. Richard Tapia and Theresa Chatman for reaching out to me and inciting me to begin and finish my graduate career.

Finally, I want to thank my family and friends for your support over the years. If not for you, much of what I have done would not have been accomplished.

Contents

Illustrations

Tables

List of Algorithms

Chapter 1

Introduction

1.1 Overview of Volumetric Data

Volumetric data, also referred to as implicit data, has been used to process three-dimensional information in a variety of industries over the recent decades. The applications for volumetric data span the biomedical, oil, computer-aided geometric design, and movie industries. From modeling solid objects to simulating fluid motion, implicit information is used to analyze a variety of real-world phenomenon. For instance, visualizing and deforming computed tomography (CT) [2, 3, 4] are relevant tasks that aid in a suite of problems from lung cancer planning treatment. Groups such as Takizawa et al. [5] generate meshes from magnetic resonance images (MRI) used in fluid simulations which determine the effectiveness of experimental treatments for aneurysms. Barges canvas oceans with miles of sensors that record three-dimensional information about the Earth's crust. Companies in the oil business need to efficiently process this data to better gauge locations for potential oil wells [6, 7, 8]. Complex machine parts are designed using Boolean combinations of solid objects in computer-aided geometric designs (CAGD) [9]. Different classes of implicit functions are used to animate popular characters such as Shrek and Toy Story's Woody in the movie industry [10, 11, 12, 13]. Interacting with volumetric data is continually addressing more problems in a variety of fields, however, with this increase in utility comes an increase in challenges.

Before we move to these challenges, we provide the following description of implicit data. Implicit data is a collection of information that represents a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$. Typical domains for f are three-dimensional volumetric data ($n = 3$) and two-dimensional images ($n = 2$.) Depending on the application, f maps to k-dimensions (usually $k \in [1, 3]$.) For example, CT images represent functions $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ where f maps a three-dimensional position to density value [2]. Deformation functions in character animation, $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, maps three-dimensional positions to new three-dimensional positions. To better illustrate the use of implicit data, we shall represent the unit circle as an implicit function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$. Given the equation $f(x, y) = x^2 + y^2 - 1$, the unit circle is defined as the set of all 2D points, (x, y) such that $f(x, y) = 0$. Referring to figure 1.1, the 2D plane is divided into two regions, inside and outside, with the level set $f(x, y) = 0$ being the boundary between the regions.

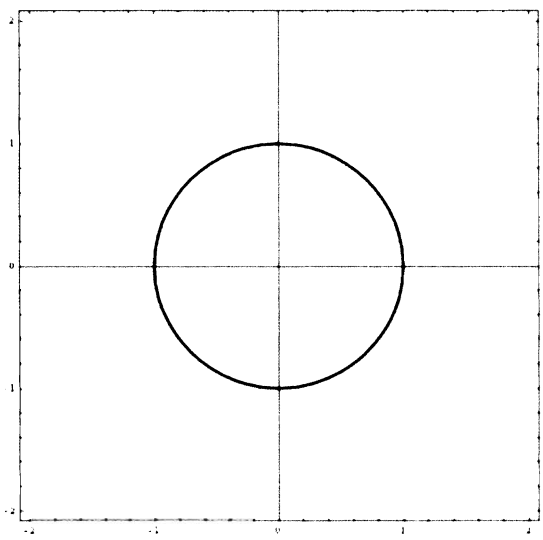


Figure 1.1 : The unit circle. Given the equation $f(x, y) = x^2 + y^2 - 1$, the unit circle is the set of all points,(x, y), on the plane where $f(x, y) = 0$

While the unit circle can be represented by a simple function, complex shapes

require more sophisticated representations. Sharp or highly smooth features of these shapes can be difficult to represent effectively with a set of equations. In these cases, groups such as Ju et al. [14, 15] approximate shapes using a function f sampled on a spatial data structure. With this information, we approximate the shape using trilinear blending on the samples. There are several spatial structures that we can use including: binary spatial partitioning (BSP) trees and octrees. For a simple introduction, we will focus on grids that sample f at regularly spaced intervals along three orthogonal axes. Here, the intersection of parallel grid lines are called grid points, line segments connecting adjacent grid points are grid edges, and the resulting regions encompassed by grid edges are grid cells. In practice, uniform grids of data come from a variety of sources such as CT scanners. To describe expressing 2D objects with implicit data, we provide the following approximation scheme:

- Determine signed distances to the surface of the shape at each grid point.
- Create an intersection point between adjacent grid points with sampled distances of opposite signs using linear interpolation.
- Connect intersection points on joined grid edges using line segments.

Figure 1.1 depicts the previous algorithm used to generate an approximation to the circle using implicit data, known as contouring. Note the effect of using grids of different resolutions; we obtain better approximations to the original shape with finer grid resolution. This is a crude method for approximating shapes, but it shows one of the primary uses of volumetric data which will be further explored later in this thesis.

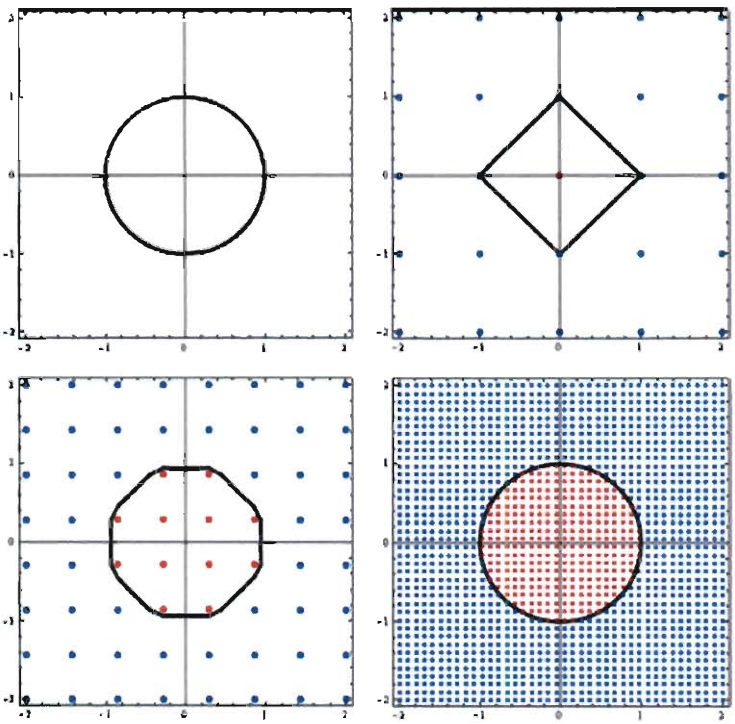


Figure 1.2 : The unit circle followed by a series of approximations to the unit circle using grids of signed distances. Red points correspond to grid points inside of the unit circle and blue correspond to outside points. As the dimensions of the grid increase, we have better approximations to the unit circle.

1.2 Viewing Volumetric Data

The most immediate use of implicit data is displaying meaningful information about the volumes. Different fields have special caveats for displaying data segmentation, cuts in the volume, and translucency. Two popular techniques for viewing 3D volumetric data are contouring surfaces from a level set of an implicit function [16, 17, 14, 15] and volume rendering [18, 19, 20]. Lorensen et al. [16] introduced the famous *Marching Cubes* method for creating surfaces from implicit data on regular grids. Subsequent work such as *Dual Contouring* [14] and *Dual Marching Cubes* [15] made substantial contributes to the field. There are many advantages with using meshes to view implicit data. Surface meshes are space efficient and can be rendered with impressive speeds with specialized graphics hardware present on commodity graphics cards.

For viewing outside boundaries of objects, surface meshes are ideal but are not well-suited for visualizing interior information. For this reason, there has been a shift toward using the recently developed features of graphics cards for storing volumetric data as textures. Figure 1.2 (left) shows volume of a human body, visualizing a portion the skin as a translucent material, but rendering the bone and internal organ structure opaque. With this recent extension, there have been a suite of techniques that render the entire contents of the volume as opposed to merely rendering a user-defined surface contour. These techniques can be divided into three main approaches: object-order [21], image-order, and domain methods. With object-order methods the contribution of each voxel to the screen pixels is calculated and the combined contribution yields the final image. An example of this method is splatting [19, 22] in which each volume voxel with properties such as color and transparency is rendered as disks from back to front order. With image-order methods such as ray casting [23, 24],

rays are cast from the screen pixels into the volume. The color of the corresponding pixel is the sum of contributions from voxels along the ray. With domain methods the spatial data is transformed into a different domain as in wavelet compression or frequency domain.

1.3 Manipulating Volumetric Data

Aside from visualizing volumetric data, altering implicit information is a crucial need in many industries. For instance, 3D artists create models using applications for sculpting and painting [14, 25, 26, 27]. These tasks are implemented as a series of Boolean operations on solid objects (see figure 1.3).

These operations include taking intersections, unions, and differences between solid primitives. For instance, let's define two circles, A and B, with radius one and respective centers at $(-\frac{1}{2}, 0)$ and $(\frac{1}{2}, 0)$. We can determine the union of the two circles, $A \cup B$, by first generating a 2D grid of signed distances from the surface and contour as previously done with the single circle example. With this information, determining the union is simply taking the minimum of the signed distances to the individual circles at each grid point. To model $A \cap \neg B$, we generate another 2D grid of signed distances where the value of f at each grid point is the $max(distancetoA, -(distancetoB))$ and contour as before. Figure 1.3 depicts these examples.

In Computer-Aided Geometric Design, software implementing more sophisticated, 3D versions of the previous sculpting algorithm generate complex machine parts used in large scale projects. Many of these models have sharp, rigid features, however, there are times where these models need to be smoothed for a variety of purposes. Smoothing models can be done by contouring the boundary mesh and running surface

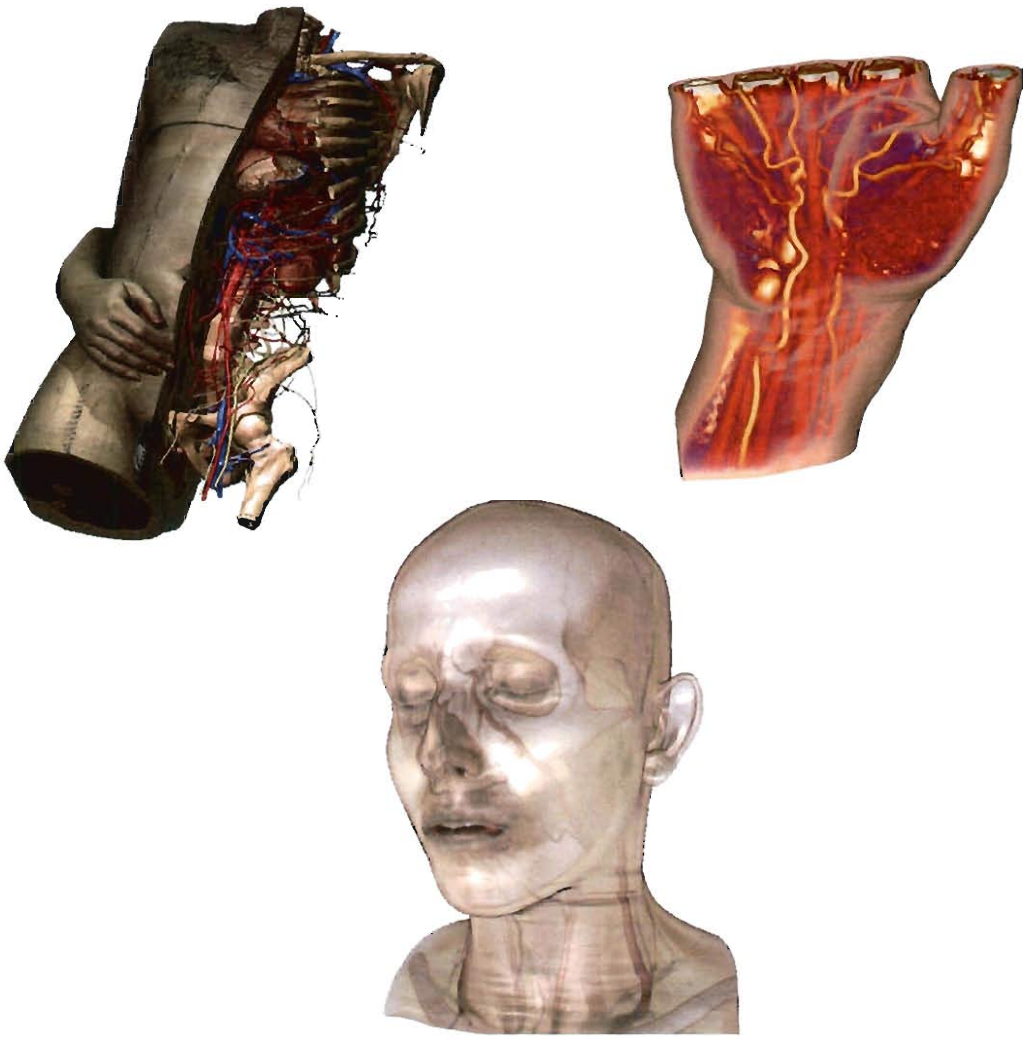


Figure 1.3 : Examples of volume visualization renderings: human torso (left), molecular structure (middle), and human head (right)

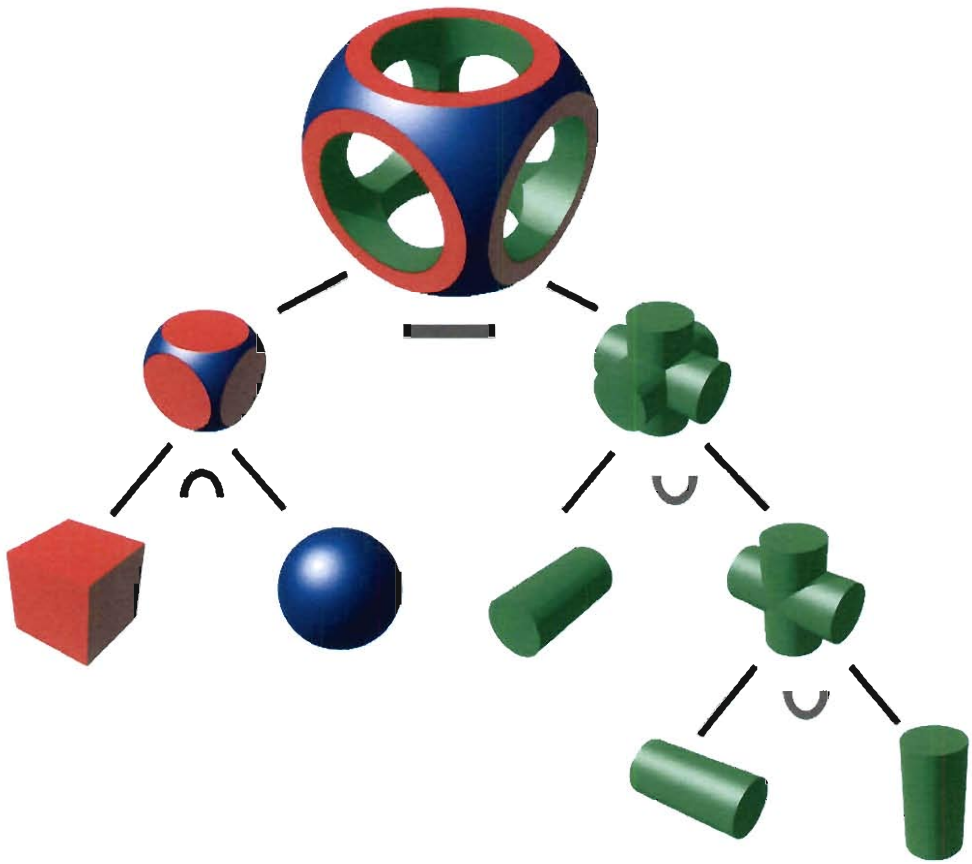


Figure 1.4 : A 3D model made from boolean operations on solid primitives. Here $\cup$ is the union operator, $\cap$ is the intersection operator, and $-$ is the subtraction operator.

smoothing algorithms such as subdivision [28, 29, 30, 31] on the boundary. But in this thesis volumetric algorithms will be discussed. Among these techniques, convolution-based algorithms and wavelet techniques [32, 33, 34] have increased in popularity. The problem of reconstructing a surface from clouds of 3D points also involves smoothing operations on volumetric data [35, 36, 37, 38].

We are not interested in merely rendering and editing models. We also care about dynamic properties such as motion. Figure 1.3 shows a caged humanoid

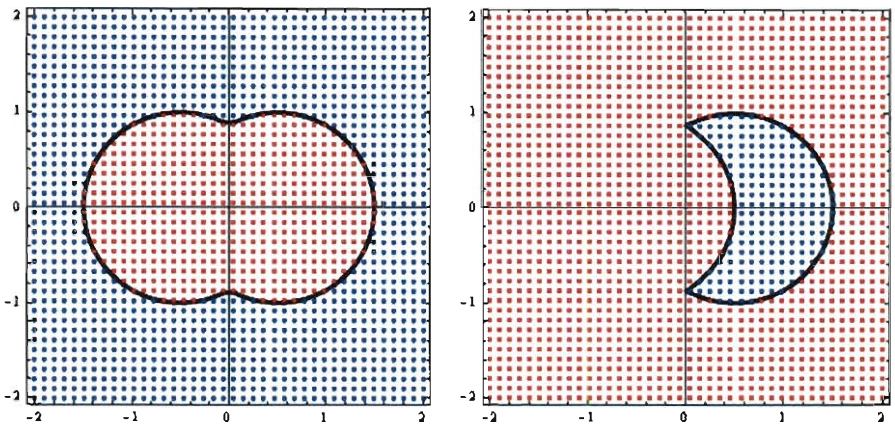


Figure 1.5 : A simple example of CSG operations on two-dimensional images. Here we have to unit circles A and B centered at $(-\frac{1}{2}, 0)$ and $(\frac{1}{2}, 0)$ respectively. $A \cup B$ is shown on the left and $A \cap ! B$ is shown on the right.

model being deformed using a skeletal controlling structure. The skeleton controls the vertices of the enclosing cage. The caged points are used as the handles for a 3D function that changes the positions of vertices on the model. In practice, these models can be quite large and problems arise with deforming them in an efficient manner. Deformation (or registration) has an extensive body of work ranging from the pure graphics community to the biomedical fields [39, 40, 11]. A large number of these approaches have undesirable features such as global effects (i.e. a small change in one region creates unintentional changes in other regions.) [11]. To form a base of understanding deformations, we move to a simple 2D deformation scheme for an image.

In figure 1.3, we have an image of a stick figure (left). To deform the image, we could simply move each pixel of the image to a new position, simulating a deformation. However, this becomes cumbersome and unrealistic for images of even moderate sizes (example: 512×512 pixels.) Instead of this approach, we turn to using a small set

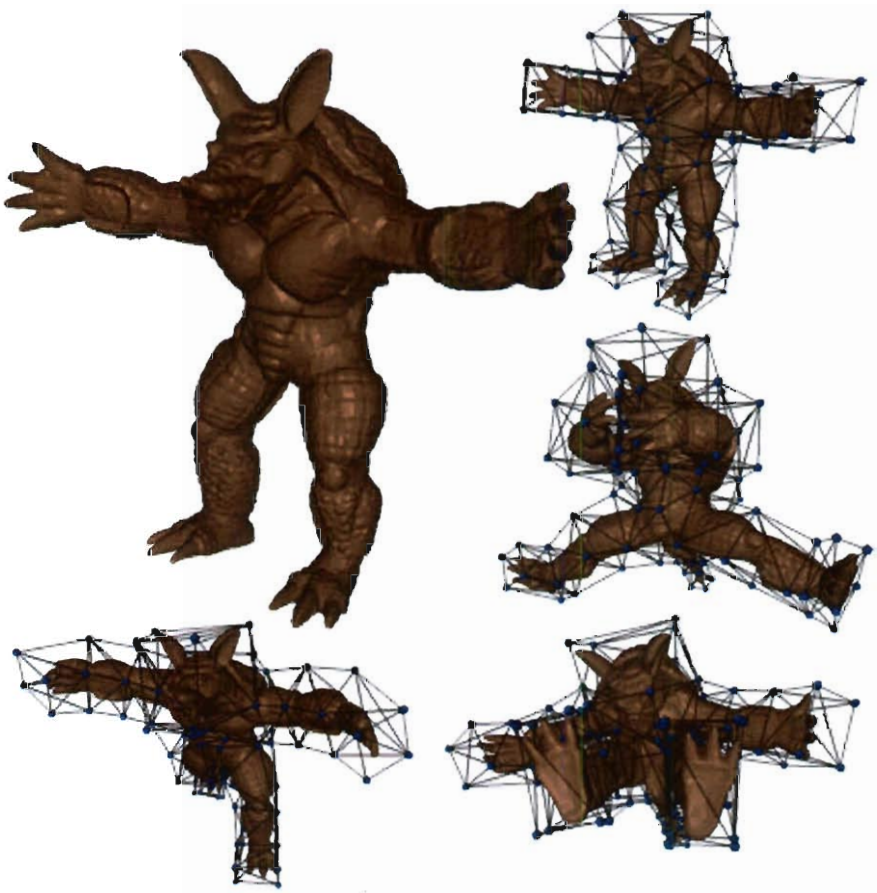


Figure 1.6 : An example of character animation. To the left we have a humanoid model which we wish to move. In the subsequent images, we embed a skeletal structure which controls the displayed cage that manipulates the 3D deformation. The final image shows a deformation induced by altering the skeletal controls.

of intuitive handles to control our deformation, the vertices of a triangle. Given the triangle, we want to represent each point on the image as a weighted combination of the three vertices $(p_1, p_2, p_3.)$ We simply represent any point v on the image as a weighted combination of the vertices, p_i, of the triangle. We implement this by generating three triangle wedges, each of which is made with v and two adjacent p_i (see the middle image of figure 1.3). With these wedges, $v = \sum_{i=1}^{3} \alpha_i p_i$ where $\alpha_i = \frac{A_i}{A_1+A_2+A_3}$. Here A_i is the area of the i^{th} wedge.

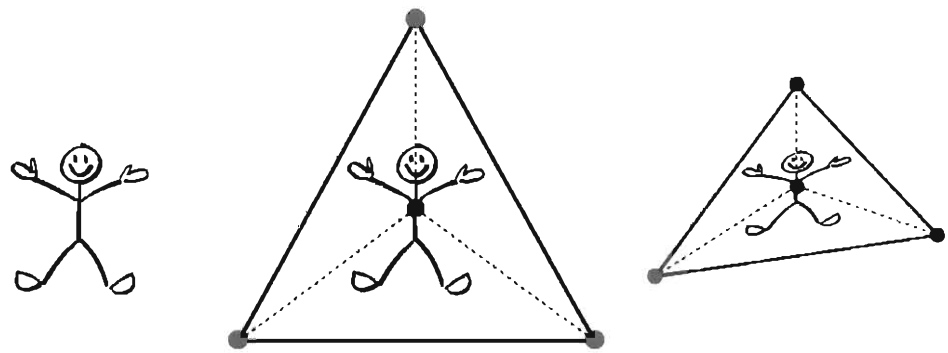


Figure 1.7 : This is a sequence depicting the deformation of a point on a stick man. The image of the stick man (left) is then encompassed by a controlling triangle (middle). In the middle image, the point v is treated as a combination of the positions of vertices of the triangle. The right image shows the deformation of v along with the entire stick man as the vertices of the controlling triangle are altered.

To apply the deformation, the user moves the triangle vertices to new positions, q_i. Now for each v in the image, we calculate a new position $v^* = \sum_{i=1}^{3} \alpha_i q_i$ (see right image in figure 1.3). While this scheme is simple, it is not expressive. These types of deformations can only represent affine transformations that are applied to the entire image. This scheme does provide a basis for work described later in this thesis.

1.4 Contributions

None of the previously mentioned areas are new and much work has been done in these fields. In this thesis, we will present novel work in the each of the following areas: rendering volumetric data (Chapter 2), realtime sculpting/painting/smoothing surfaces using volumetric techniques(Chapter 3), deformation of volumetric data (Chapter 4), and an application utilizing this deformation to address lung cancer treatment planning (Chapter 5).

In chapter 2 we add an improvement to a portion of the rendering pipeline for texture-based volume rendering approaches. Texture-based methods use a sequence of polygons parallel to the user's perspective to render 3D volumetric information. These polygons are generated from the intersection of a plane sequence and the unit cube. We present an approach that uses a table that maps a view vector to a sequence of topology changes for the intersection. We also provide a sweeping process to efficiently generate the sequence of polygons.

In chapter 3 we present an editing environment that allows a user to perform CSG operations using arbitrary closed mesh models. This framework utilizes depth peeling and GPU advances to reflect these computations in realtime. After the user finishes with a set of edits, a surface is contoured only in regions that have been altered. This altered surface is then patched to the untouched original surface. This framework facilitates painting and a novel smoothing feature that utilizes a semi-volumetric representation. One of the main advantages of this work is performing computations on a structure that is proportional to the surface area of the utilized meshes as opposed to being proportional to the respective bounded volumes.

Chapter 4 presents an intuitive framework for generating believable image deformations using simple handles, points and lines. We are able to implement a range of

deformations: affine transformations, similarity transformations, rigid deformations, etc. With a small precomputation overhead, these deformations can be determined in realtime.

Finally chapter 5 utilizes the above deformation technique in an environment for registering three-dimensional images. We focus on applying this environment to the realm of lung cancer planning treatment. We also present a method for semi-automatic segmentation of lung lobes and give experimental results relating 3D deformations of breath-hold CT images with lung ventilation.

Chapter 2

Fast Cube Cutting for Interactive Volume Visualization

Comprehending the geometric and physiological content of volume data has applications in a variety of fields: medical imaging, geoscience, astrophysics, molecular visualization, and fluid dynamics. There are two primary means for visualizing volume data: polygon rendering (or indirect volume rendering) and direct volume rendering. Polygon rendering involves extracting a surface mesh from grid data and using the conventional rendering pipeline to display the surface. These meshes correspond to level-sets of the implicit function given by the volume data. Most well-known works in this area are Marching Cubes and, more recently, Dual Contouring [41, 42]. Direct volume rendering (DVR) is the class of techniques that draws volumetric data directly without constructing polygons that approximate the data [43]. DVR methods can provide internal structure information by using techniques such as transfer functions. This extends beyond the capabilities of typical indirect volume rendering methods.

2.1 Related Work

In the history of direct volume rendering techniques, ray-casting, shear-warp, splatting, and texture mapping stand out as being most used and researched [22]. Raycasting and splatting techniques are high-quality but non-interactive methods, whereas shear-warp and texture mapping are fast, hardware-supported methods but lack in fidelity without extra care [44, 45, 46, 47]. We will focus on texture-mapping techniques

in our work.

Because of the wide availability of hardware support, texture-hardware-based volume rendering has been a popular topic for research. The technique of texture-mapping requires generating a set of view- or object-aligned proxy geometries. These geometries are assigned texture coordinates at their vertices, and texture-mapping from these coordinates is either done on the hardware or the software. Cullip et al., Wilson et al., and Cabral et al. were among the first to suggest the use of texture-mapping hardware with volume rendering [47, 48, 49]. Much of the earlier work focused on using object-aligned 2D-textures. Lacroute and Levoy used object-aligned slices to render volumes, but they did not rely on hardware acceleration for rendering [46]. Westermann and Ertl described methods for drawing shaded iso-surface using OpenGL extensions to exploit 3D texturing capabilities on the hardware [50]. Rezk-Salama et al. described a method for shaded iso-surfaces using 2D textures [51]. Engel et al. improved the quality of texture-based rendering with a pre-integration technique [52]. Other works on the subject span a wide range of techniques that leveraged performance against quality of rendering [53, 54, 55].

2.2 Contributions

In this chapter we address the problem of generating proxy geometries in texture-hardware-based volume rendering. In general, this involves trimming a set of view-aligned planes to a cube, as illustrated in Figure 2.1. There are several works that are related in this respect. In particular, Dietrich et al. proposed a plane-sweep algorithm for plane-cube intersection that is similar to our work [56]. However, their description is brief, their application is different, and it is unclear how geometries are generated in their algorithm. Rezk-Salama and Kolb have described a method for computing proxy

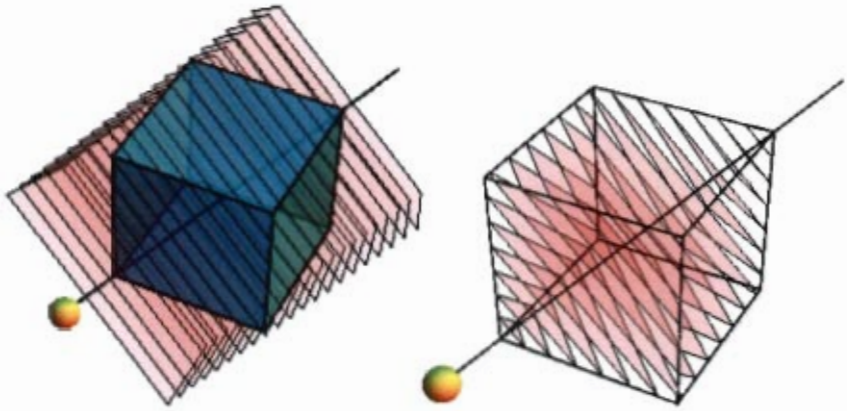


Figure 2.1 : Comparison between series of quadrilaterals (left) that span the volume of interest versus polygons trimmed to the exact boundary (right). In the left figure, note the amount of superfluous texels that do not contribute to the image.

geometries on the vertex shader, where each plane/cube intersection is computed independently [1]. Additionally, six vertices are passed to the GPU for every proxy plane, which is redundant in many cases. In contrast, our method pre-computes all unique sequences of topology changes from plane/cube intersections and stores the information in a table. Next, we develop a sweeping algorithm that efficiently computes the exact polygonal intersection between the topology changes given by our table. Furthermore, we explore this algorithm in an octree-based empty-space culling framework. We will compare our method against Rezk-Salama and Kolb's work to show that a plane-sweeping method is more efficient in the case of multiple plane-cube intersections. Furthermore, we also compare with the use of hardware clipping planes and show that our method performs well in comparison.

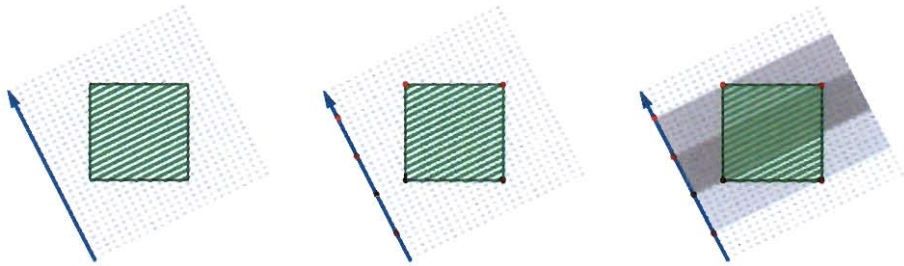


Figure 2.2 : Illustration of wedges with respect a view vector. We start with a view vector and cube (left). We project the vertices on the view vector(middle). Then we define the regions between the projected vertices as *wedges*.

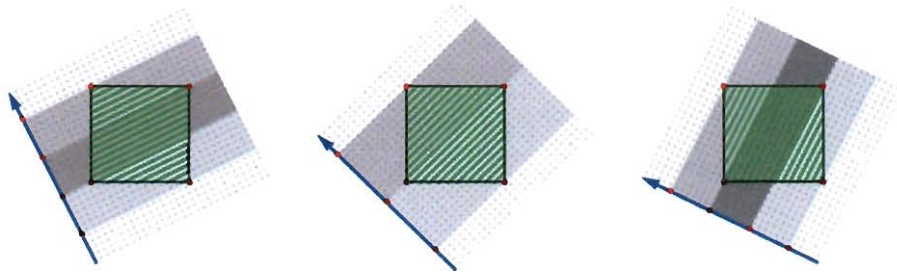


Figure 2.3 : Series of images that illustrate the occurrence of a change in the order of the vertices. The middle image shows that two vertices have the same projection on the view vector when the ray between them is perpendicular to the view vector.

2.3 Method

2.3.1 Overview

As noted in the introduction, the key to texture-based hardware rendering of 3D volumetric data is generating a set of proxy geometries for mapping volume densities into 3D rendering space. Typically, this process involves generating a set of intersections between a single cube and a sequence of parallel planes. These planes are usually perpendicular to a given viewing vector and equally-spaced. These proxy geometries tile the volume so that the composite of their texture-mapped images produce a 3D rendering of the data. Figure 2.2 and Figure 2.1 shows 2D and 3D examples of this tiling processing, respectively.

This chapter is built on two key observations. The first observation is that, given a fixed view vector, the cube can be partitioned into a finite set of *wedges*. These wedges are formed by projecting the vertices of the cube onto the view vector and building a set of planes perpendicular to the view vector that pass through these projected points. Figure 2.2 illustrates this process in 2D.

Inside a given wedge, any plane perpendicular to the view vector intersects the cube in the same set of cube edges. If one wishes to sweep this plane through the wedge, the only updates that need to be done are incremental updates of the positions of the edge intersection points. (Note that no topology calculations need be done.) Thus, the intersection of the cube and a swept sequence of planes perpendicular to the view vector can be computed extremely efficiently inside a single wedge.

More generally, if one could pre-compute the set of edge intersections of each wedge, a simple plane sweep algorithm could be used to tile the entire cube. As a plane normal to the view vector is swept through the cube, an outer loop tests

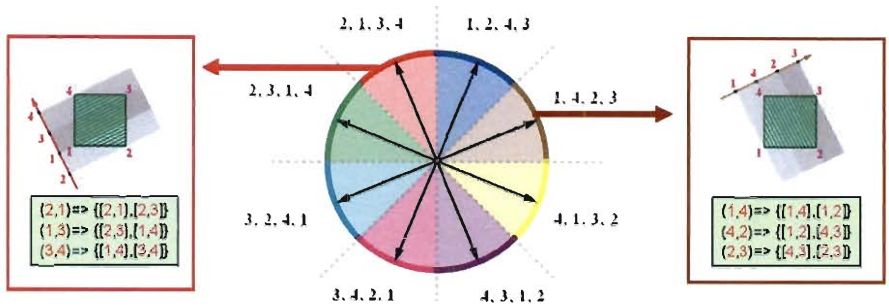


Figure 2.4 : 2D depiction of the partition of the view vectors for a 2D cube and two sample sequences of topological changes (STC). The STC is represented as a list of vertex orderings followed by an edge list.

whether the plane had moved from one wedge to the next wedge. If not, a second inner loop continues the sweep through a given wedge.

The second key observation is that varying the view vector slightly normally does not affect the algorithm described above. The set of intersected edges associated with each wedge remains unchanged unless *the order of the projected vertices changes.* Figure 2.3 shows an example of three different vertex orderings and their associated sequence of wedges for the 2D case.

Our approach will be to pre-compute a *sequence of topology changes* (STC) for each possible distinct ordering of projected vertices. An STC will be indexed by its vertex ordering and consist of a list of edge intersection sets associated with each wedge. In the 2D case, there are only eight distinct STCs associated with all possible view vectors. Figure 2.4 depicts these eight case and the STCs associated with two of these cases. This approach is viable in 3D since the numbers of distinct vertex orderings is also relatively small (96) (see Figure 2.5 for the 3D partition of the view vector space).

We will briefly describe how to apply our method. The table containing STCs is invariant of the rendering parameters. As long as our bounding polyhedron is a cube, we only need to compute this table one time and use it to supplement the texture-based volume rendering algorithm. During the rendering phase, for every change in view direction, the user queries the table of STCs using the view vector. The query returns an STC for the current view direction. Using this STC, the user then applies the plane sweep algorithm, which outputs a set of cube-trimmed polygons. These polygons are sent to the graphics card for 3D volume texturing using standard texture-based DVR. Hence, our method relies on CPU trimming to relieve the GPU of excessive processing.

In the next two subsections, we discuss:

- A method for constructing the table of STCs associated with each distinct vertex ordering as well as a fast method for determining vertex ordering,
- The resulting plane sweep algorithm for generating trimmed polygons.

2.3.2 Table Structure

Our motivation for building a look-up table for STCs is based on the observation that the number of all vertex orderings is small. Akin to the sign-change table in Marching Cubes, we use table look up to avoid the need to do topological calculations while generating the intersection of the swept planes and the cube.

As noted earlier, the look-up table has one entry for each distinct ordering of the projected cube vertices. A naive approach would be to consider all 8! possible vertex orderings of the eight vertices of the cube. However, this approach is unwieldy. In practice, most of the vertex orderings are impossible. For example, in Figure 2.4, the

of the cube, construct its perpendicular bisecting plane P_{ij},

- Use these planes P_{ij} to partition the unit sphere centered at the origin into a set of spherical patches S_k. (Note that there are multiple copies of the same bisecting planes.)

- Points on each patch S_k correspond to the set of view vectors for which the ordering of the projected cube vertices is the same. For each patch S_k, compute this vertex ordering and its associated STC.

Figure 2.4 shows the partition of the unit circle into eight circular arcs by the four distinct perpendicular bisecting planes associated with the square. In 3D, there are 13 bisecting planes; 3 distinct planes for edge adjacent vertices, 6 distinct planes for face adjacent vertices and 4 distinct planes for diagonally opposite pairs of vertices. These planes then partition the unit sphere into 96 spherical patches shown in Figure 2.5.

To complete our table lookup method, we need to construct a method that, given the view vector, computes the index k for the patch S_k. Our method is a simple one based on BSP trees [57]. Given a point on the unit sphere, we can perform a sequence of front and back tests against the 13 planes reference above to determine which spherical patch the point resides in. We associate a distinct plane at each level of the BSP tree. So for the 2^n nodes on level n of the tree, the same plane is used to perform our front/back test. So at each node of the tree if the point is behind the plane, we continue with the left child of the tree and the right otherwise. At the leaves of the BSP tree we store the sequence of topology changes of the intersection of the cube with the sweeping plane associated with the provided view vector. The order of the plane tests does not affect the results, but we choose the planes for our BSP tree using the following observations. First we note that the 3 distinct bisecting planes

associated with edge adjacent cube vertices partition the unit sphere into 8 octants. (One octant is highlighted in Figure 2.5). Given a view vector, the top three levels of our BSP tree use these three planes to split the unit sphere into eight octants. Next, note that the six distinct bisecting planes associated with the face adjacent pairs of vertices partition this octant into six triangular patches. (In fact, only three planes even intersect the patch in the dashed edges). The next three levels of the BSP tree use these three planes and partition each octant into six triangular patches. Finally, we note that only one of the four distinct bisecting planes associated with diagonally opposite pairs of cube vertices intersects a particular triangular patch. The last level of our BSP tree uses this plane to partition this triangular patch into two patches, one of which is our desired spherical patch S_k.

Thus, this BSP tree of depth seven allows one to compute the vertex ordering and associated STC using only seven dot products. Given this information, we now present our plane sweep method in detail.

2.3.3 Plane Sweeping

Given a view vector v, we want to generate a set of proxy geometries, which are the intersection of view parallel planes and a cube. Our algorithm utilizes the BSP-tabularized result given in the previous section for fast look-up of the topological changes with respect to v. Also stored within the BSP tree is the order of the cube vertices projected onto the view vector.

We compute the plane-cube intersection by starting from one end of the cube and sweep through the cube in the direction of the view vector. We use P to denote the current plane of the sweep, and P is always normal to the view vector. Let d be the step size of the plane sweep; this is a user given parameter that determines the

denote the plane's advancement). While P is still in W_k, we know that all trimmed geometries will have the same topology. When P crosses over into W_{k+1}, then we need to again find the starting vertices for T_{k+1}. (Instances of where this computation is necessary are marked as red circles in Figure 2.6). This process iterates until P crosses the last wedge. The pseudo-code is given in Figure 2.6.

The running time of this algorithm is linear in the number planes specified by the user. We cut down on the constant factor of the asymptotic running time by using a table look-up and by replacing edge-plane intersections with vector additions. The initializing step for each wedge presents the heaviest load of computation. However, this step only occurs a small number of times in a sweep.

2.4 Results

We ran our method against two other methods for comparison. The first is the software implementation of Rezk-Salama and Kolb's method (denoted as RSK) [1]. The second method for comparison uses hardware clipping planes to trim the cubes; this is a straightforward GPU implementation with minimal computation on the CPU. We include results from volume-rendering with no plane trimming (No Op) to establish a baseline of comparison. This is a naive method that sends untrimmed, view-perpendicular, and screen-filling quads to the graphics card (See left image of Figure 2.1). Although the "No Op" method incurs the very little CPU computation, it introduces wasteful pixel processing for empty regions.

Our primary test machine is a Pentium Xeon machine with two 2.66GHz CPUs and 4GB of memory. The graphics card is NVIDIA Geforce 8800 GTX with 768MB of texture memory.

We compare our method against RSK by counting just the number of planes

that can be trimmed in a fixed amount of time. RSK showed a speed of 650,000 planes trimmed per second, and our method posted a speed of 2,500,000 planes per second. Although our method is 3 to 4 times faster than RSK, the empirical rendering results we gathered only indicate marginal speed-up. This is due to the fact that the rendering time is mostly dominated by filling pixels as opposed to plane-cube intersections.

2.4.1 Single Cube

The results from Figure 2.7 suggest that all three methods perform two to three times better than volume rendering without any optimization. First, for the 500×500 tests, our method has higher frames-per-second than the other two methods. We observe that as the rendering resolution increases from 500×500 to 1000×1000, pixel fills becomes the bottleneck.

We ran tests on a second machine to establish the trade-off between CPU vs. GPU in terms of hardware differences. Our second test machine is a Pentium 4 machine running at 2.8GHZ with 1GB of memory. Its graphics card is NVIDIA Geforce 6200 with 256MB of texture memory. In Figure 2.8, the hardware clipping-plane method has the lowest number of frames-per-second. However, its performance in the high-end machine tests is highly competitive. Given this trend, we believe that as more powerful graphics hardware become available, the clipping-planes method can outperform either of the two software implementations on a desktop machine. In the next section, we will discuss a case where the performance issue cannot be addressed with better hardware.

this quick polygon generating approach in an octree-based empty-space culling framework. We have shown results that indicate our method is more efficient than previous methods, and in the case of empty-space culling, it outperforms the pure hardware implementation.

For future work, we would like to examine moving some of the computation on to the graphics hardware. As implied in the results section, load-balancing between CPU and GPU requires careful analysis; furthermore, this process is highly dependent on the hardware specification. We think that a careful implementation can make use of the increasingly faster GPU and transfer some of the work-load to the GPU. This direction is especially relevant given the recent development of geometry shaders.

Chapter 3

Depth Peel Editing

We now focus on using an implicit approach to surface editing. Implicit modeling is used in a variety of areas such as: architecture, character modeling, biomedical imaging, and milling. Within implicit modeling, generating intricate polygonal meshes by molding primitive objects is common practice. Typically this molding is defined by a combination of Boolean operations (unions, intersections, and differences) performed on the input primitives.

Current modeling tools (e.g. ACIS, 3D Max, Parasolid, and Maya) primarily use simple polygonal meshes or shapes represented as real functions over a 3D domain as inputs to Boolean expressions due to the compact representation and hardware-accelerated rendering capabilities. These modeling applications can model boolean operations on complex shapes, however, when computing such boolean operations directly on complex surfaces, issues with numerical stability arise [58, 59, 60]. Additionally, difficult logic is needed to perform all possible intersection cases between faces, edges, vertices. To avoid these problems the general approach in many applications is to convert the polygonal mesh to a volumetric representation. In this conversion, commonly known as 3D scan-conversion, the surface is approximated by a number of volume elements (i.e. voxels.) Subsequent boolean algorithms are applied to the volumetric representation, which is more robust and a final surface is extracted from the volumetric result [16, 14].

3.1 Related Work

The evaluation of Boolean expressions is accomplished using the intuitive tree-traversal method known as Constructive Solid Geometry (CSG) [61, 62, 25, 60]. CSG expressions are described by a binary tree of corresponding Boolean operations on child CSG expressions. The leaf nodes of CSG trees are the solid primitives. As previously stated, many applications incorporate an initial 3D-scan conversion of input meshes on to either uniform or adaptive volume grids which can be a bottleneck in computation. There has been a great deal of work on 3D-scan conversion, or voxelization [63, 64, 65, 66]. While Eisemann et al. [65, 66] utilize graphics hardware to perform voxelization, the resulting sizes of these grids is on the order of n^3 where n is the resolution of the grid. On average a substantial portion of these grids represent empty data, which yields many unnecessary computations and inefficient storage.

Current work with CSG expressions focuses on rendering CSG trees with many leaf primitives [67, 68, 69, 70, 71]. [69] introduces a GPU-based method for rendering CSG trees. The authors combine depth peeling [72] and their Blist formulation to render arbitrary CSG models. [71] and [70] incorporate the use of an octree structure to prune empty regions in the CSG rendering, but they only present convex primitives.

The major drawback of these approaches is their sub-interactive rendering times due to the number of primitives rendered simultaneously or limited complexity of the CSG primitives. As opposed to rendering CSG trees with simple objects, we wish to render sessions where users construct a CSG tree at interactive speeds using complex polygonal meshes. Furthermore, we wish to generate a polygonal mesh from the CSG tree the user has defined. In all of these works, information from the CSG operations is not retained which makes subsequent mesh generation infeasible.

Substantial progress has also been made on contouring Boolean operations be-

tween polygonal meshes [73, 74]. [73] uses 3-Layer Depth-Normal Images (based on depth peeling [72]) to create an adaptive grid of hermite data which is used to generate a surface via a Extended Dual Contouring [75] approach to generate the resulting surface. While this is a novel approach, it contours the entire volume as opposed to restricting contouring to critical regions. The 3-Layer Depth-Normal representation is near identical to the hybrid representation used in this chapter, but the works were done independently. [74] generates an octree of hybrid information which contains inside/outside information for each grid point directly from CPU operations on the input meshes. With this octree representation they restrict contouring to "critical" cells (i.e. cells that contain portions of multiple surfaces). They also provide an algorithm for patching the resulting contoured surface with the original primitive surfaces. In both approaches, timing results are on the order of seconds which we wish to improve to milliseconds.

3.2 Contributions

In this chapter we discuss a modeling framework that is a hybrid between surface and volumetric representation which takes advantage of the strengths of both approaches. Our methods enable interactive, view-dependent sculpting and painting of closed surfaces using arbitrary meshes as modeling tools. At the core of the framework, we generate a hybrid volumetric representation of polyhedral meshes which is proportional in size to the original mesh as opposed to performing full 3D-scan conversion on the input meshes. We also perform modeling operations based on this hybrid representation. This enables us to improve the runtime of these operations from $O(n^3)$ to $O(n^2)$ where n is the dimensions of the sampling grid. We also create an algorithm for performing local smoothing operations on a surface mesh within a

user-defined window. Our algorithm reconciles the smoothed surface with the exterior unaltered surface. Additionally, we render these modeling operations at near realtime rates allowing the user to edit polygonal meshes at interactive speeds. Furthermore we adapt Dual Contouring [14] on our volumetric representation to generate new surface features restricted to the edited regions of the surface. Finally, we provide a stitching algorithm that patches the edited surface with the original shape.

3.3 View-Dependent Depth Peel Representation

In 2001, Everett et al. [72] introduced a technique for viewing transparent objects without pre-sorting the polygons according to decreasing depth value *depth peeling*. Given a scene of surfaces meshes, depth peeling repetitively renders the scene elements generating a sequence 2D *depth layer* images at screen resolution. The resulting image is the composition of this sequence of images. Depth peeling can be understood from examining a ray originating from the viewer's eye and passing through a screen pixel much like raycasting [76, 77] and shadowmapping [78]. As a screen ray passes through pixel (sx, sy) and intersects surfaces in the scene, properties such as reflected color and translucency are stored for each intersection point in a corresponding depth layer image at position (sx, sy). These intersection points are stored in order sorted by closest depth value, so information for the i^{th} closest intersection point is stored in the i^{th} depth layer. In a scene of depth complexity [79, 80] N, if a screen ray has M intersection points where $M < N$, the background color is stored at (sx, sy) for the last $N - M$ depth images.

directions with corresponding 2D grid of rays R_x, R_y, and R_z using orthographic projections. The views are constructed in such a way that the intersection of the rays ($R_x \times R_y \times R_z$) form a uniform grid in $\mathbb{R}^3$. These three views provide information along all three orthogonal directions in the uniform grid yielding more accurate contour results.

3.3.2 Depth Peeling Image Implementation

Our major goal is to create a framework for interactive mesh editing using surface meshes as editing tools. In such an environment, it is intuitive for a user to move an object to a desired view, perform a sequence of mesh edits then continue viewing the altered surface. To accommodate, we render the actual surface while the user alters the view of the mesh, but freeze the frame buffer when the user begins edits and alter pixels as the user performs implicit operations. Figure 3.2 describes the organization of computation during the editing process. While the user alters the view on the base surface mesh, our framework utilizes basic GPU functionality for surface rendering. Once the user freezes the view to begin edits, we generate four DPIs of the base surface mesh (one taken from the perspective view of the user and three taken from cardinal orthographic views.) In practice the user only interacts with the perspective view. Generating a DPI involves multiple renderings of the target surface mesh and passing depth, color, and normal information from the GPU memory to the host memory during each rendering pass. Copying the contents of the framebuffer during each rendering is the major runtime bottleneck for generating a DPI, however, experimental runtimes are still on the order of tens of milliseconds. All DPIs are managed on the CPU with the perspective view managed on the main application thread and the orthographic views managed on a background thread. A

DPI generated from a view direction is represented as a 2D array of list of surface normal/depth/material tuples.

As the user performs subtraction/addition operations with an editing mesh, we generate a DPI of the edit mesh with respect to the user's frozen perspective. The time needed to generate the DPI depends on the convexity of the edit mesh; but since the DPI is generated using the GPU, the time is on the order of milliseconds. Afterwards, we merge the edit DPI with the base DPI (described in the next section). The merging process is reflected for the three orthogonal DPIs on a separate thread. Furthermore, we determine a bounding box for the altered screen pixels. Then for each altered pixel, we traverse the respective list of ray points and display the color of the first intersection. Note that we could implement transparent materials by simply iterating through the list of intersection points. These caveats allow users to interactively mold complex surfaces with non-trivial mesh tools.

Our sequence of three DPIs provides a robust and compact implicit representation for surface meshes. In order to faithfully contour this implicit form, we need to ensure consistent material classification for every point on the grid. However due to infrequent numerical precision issues the material assignment for a point may not agree in the three DPIs. For example if a piece of surface is close a grid point, two DPIs might classify the point as outside while the third classifies the point as inside. We rectify this by enforcing a majority vote. If two or more DPIs gives a grid point *Material A* then the grid point has *Material A*.

3.4 View-Dependent Editing

Using the frame buffer to reflect surface operations yields highly accurate results which can have intricate details from the editing mesh. If we embed our operations in

a fixed uniform grid, it becomes difficult to contour surfaces that accurately capture intricate features as a user zooms in on the base mesh. For example if we take the popular Stanford dragon model, zoom in on one of its eyes, and carve out the Stanford bunny we would want the detail of the bunny model to be preserved in our contouring phase. We take a view-dependent approach to address this issue. The pseudocode for generating the bounding box follows:

Algorithm 3.1 Computing bounding box for surface editing

Require: $\delta_{min} \rightarrow$ minimum projected z-value from depth buffer

Require: $\delta_{max} \rightarrow$ maximum projected z-value from depth buffer

Ensure: Determine Bounding Box

1: Determine δ_{min} and δ_{max} from user view with depth buffer

2: Generate spherical $plane_{min}$ and $plane_{max}$ by shooting rays of length δ_{min} and δ_{max} from the user eye

3: Determine extrema for the planes as (low_x, low_y, low_z) and $(high_x, high_y, high_z)$

4: Set $axis_{length} = max(high_x - low_x, high_y - low_y, high_z - low_z)$

5: Set mid to the midpoint of the extrema

6: Set $box = (mid_x, mid_y, mid_z) \pm (\frac{axis_{length}}{2}, \frac{axis_{length}}{2}, \frac{axis_{length}}{2})$

When a user freezes the view of the base mesh, we find the tightest bounding box for the visible features by unprojecting screen x, y, depth values returned from the depth buffer. We further pad the cube in order to create a reasonable bound for surface additions. Afterwards we apply proper view transformations so that the intersection of rays from the three orthographic DPIs infer a uniform grid at screen resolution within the bounding box from algorithm 3.1. Figure 3.3 gives 2D examples of bounding boxes for different views of a scene. For the duration of this chapter, we

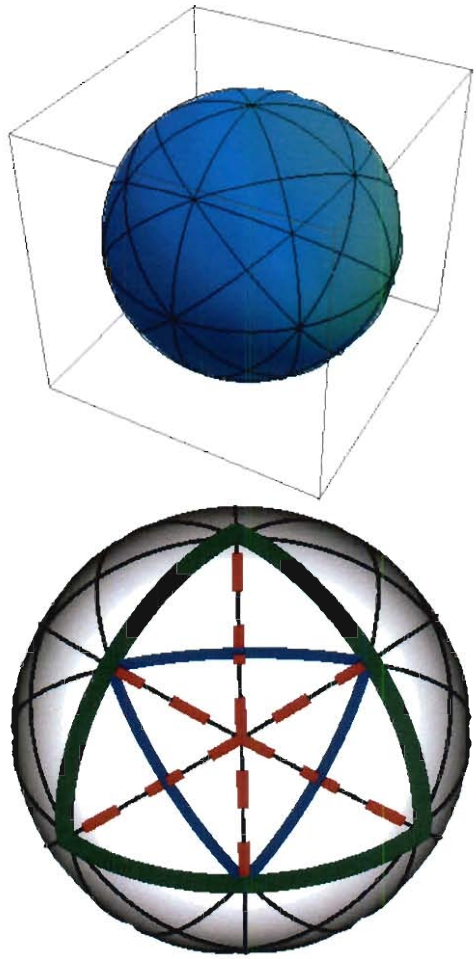
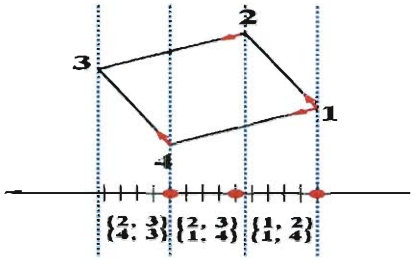


Figure 2.5 : Partition of the view vectors for a 3D cube. The highlighted portion of the right figure represents a octant.

ordering $\langle 4, 2, 1, 3 \rangle$ is impossible.

A more precise method for constructing the set of possible vertex orderings is to note that the two vertices of the cube project onto the same point on the view vector if their perpendicular bisecting plane is normal to the view vector. This observation leads to the following algorithm for constructing the look-up table.

- Given a unit cube centered at the origin, for each pair of distinct vertices c_i, c_j




```
k ← 1
P ← plane containing cube vertex closest to eye
while k ≤ N
    ∀(i,j) ∈ T_k, m_ij ← intersect P with edge (c_i, c_j)
    while P is in wedge W_k
        ∀(i,j) ∈ T_k, m_ij ← m_ij + y_ij
        P ← advance P in the direction of v by d
    increment k
```

Figure 2.6 : The left figure is an example of the plane-sweep algorithm in 2D. The bottom figure shows the pseudo-code of the algorithm. The notations are detailed in the following section.

number of trimmed geometries. We let the wedges be defined as $W_1, W_2, \ldots, W_N$, and note that the number of wedges varies with respect to the view vector (wedges are the regions between dotted blue lines in Figure 2.6). Let $T_1, T_2, \ldots, T_N$ be the topology (a set of edges) associated with each wedge. (In Figure 2.6 the topology of the first wedge is $\{\{1,2\}\{1,4\}\}$). We write $u_{ij} = c_i - c_j$ where (c_i, c_j) is an edge in T_k. (These vectors are oriented in the same direction as v; in Figure 2.6, they are the red arrows).

The algorithm proceeds thus: we initialize P to contain the vertex closest to the eye point. Then we step through each wedge W_k, where for each edge, we compute the intersection of P with the edges of T_k. These intersections are denoted as m_{ij} for each edge $(c_i, c_j) \in T_k$. These intersections will be the starting vertices for the current wedge. After finding the starting vertices, we generate the coordinates of the following trimmed geometry by vector addition. We add to the geometries of the previous iteration by the scaled vector

$$y_{ij} = \frac{d}{u_{ij} \cdot v} u_{ij}$$

for each of the edges in T_k. We also advance P by a fixed distance d along the view vector in every iteration when generating the new geometries (tick marks in Figure 2.6

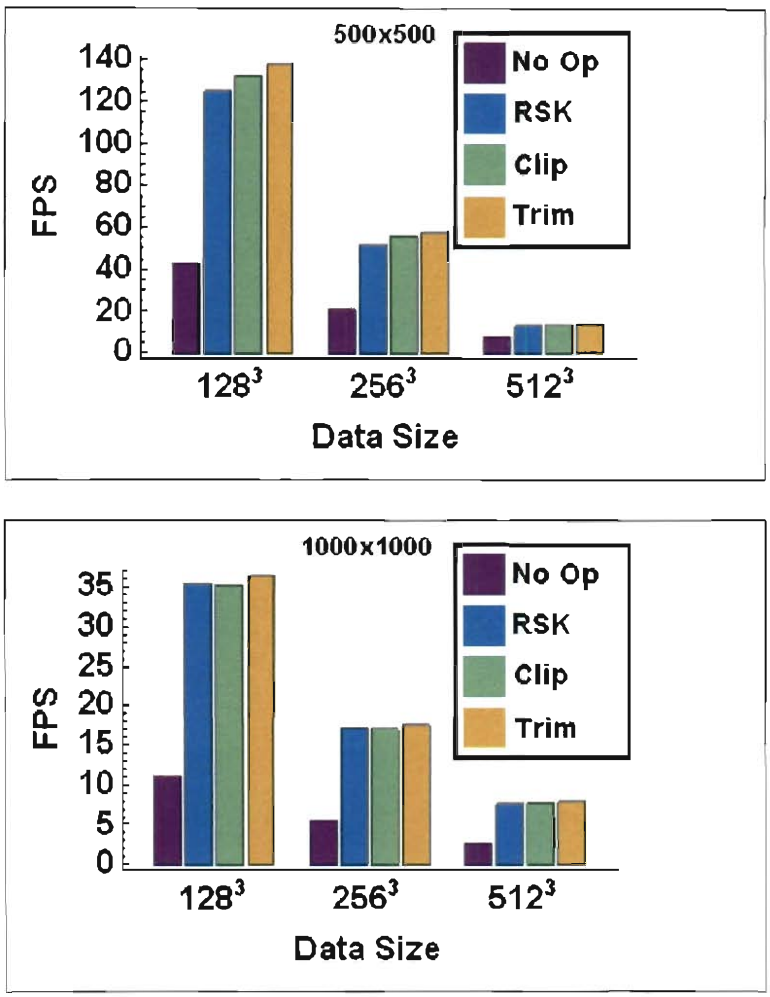


Figure 2.7 : The top graph records the frames-per-second for each method where the rendering window is 500 × 500 pixels. The bottom graph has results for 1000 × 1000 pixels. "No Op" stands for volume rendering with no plane-trimming; "RSK" stands for the method by Rezk-Salama and Kolb as described in [1]; "Clip" stands for the hardware-clipping-planes method; and "Trim" is our cube-trimming algorithm.

2.4.2 Empty-Space Culling using Octree

We use a simple heuristic for empty-space culling by decomposing the volume into an octree where each cell stores the min and max values of the contained subvolume.

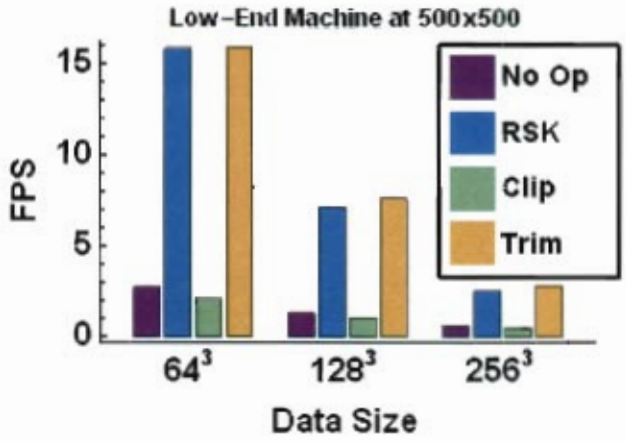


Figure 2.8 : Results of all the methods running on a machine with limited graphics hardware.

This is a simple scheme based on ideas from Lamar et al [54]. With the optimization, we can avoid rendering regions that are not within our range of interest. We apply the three methods on each octcell and render the resulting planes or polygons.

We used a largely empty volume as our test example. The results of Figure 2.9 indicate that empty-space skipping can provide significant speed-up. However, we see the gain from performing empty-space skipping by octree-decomposition diminishes as the octree traversal depth increases beyond level 4. This result is expected because the decomposition at level 4 is refined enough such that the waste due to rendering empty-spaces is minimized.

In Figure 2.9, we see a drop in frame rate for the clipping-plane method for levels beyond 2. This is due to the high number planes that have to be passed to and trimmed by the GPU. Since the GPU has to handle both the volume clipping and rendering, its performance staggers as the number of planes increases. We conclude that the hardware approach is not suitable for cases with high number of planes, such is the case with empty-space culling using octrees. Again, note that our cube-

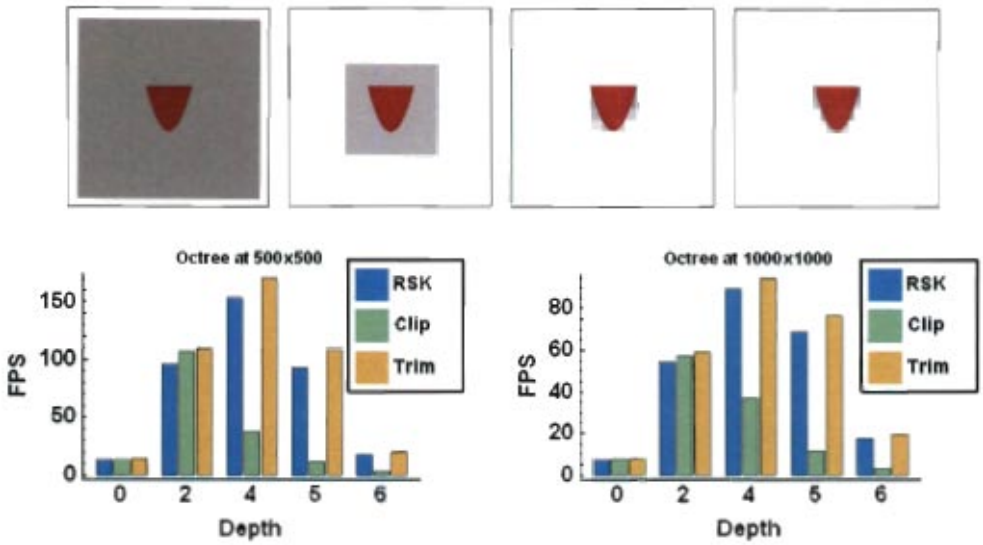


Figure 2.9 : Results of the three methods running with octree decomposition enabled. "Depth" represents the depth of the octree decomposition. The top four figures show the top-down views of the largely empty volume used in this test. The gray box is the bounding octcell in the octree. The four different levels of decompositions are 0, 2, 4, and 5 (from left to right).

trimming method outperforms RSK.

2.5 Conclusion

We have provided a method for generating the intersection of multiple planes defined by a view vector with cubes. For a given vector, we use a sweeping algorithm that uses the vertex connectivity of a cube to create a sequence of topology changes for the plane as it moves through the cube. We tabularize all possible sequences of topology changes for a given cube into a BSP tree. We use this table to quickly map a view vector to a sequence of topology changes and perform a simple sweeping algorithm that generates the polygonal intersections via simple vector addition which is computationally more efficient than regular plane-edge intersection tests. We use

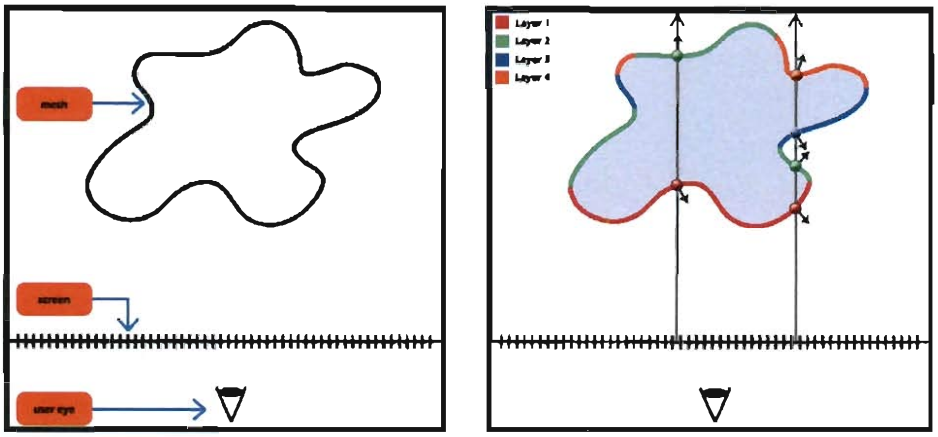


Figure 3.1 : Pair of images demonstrating depth peeling as a volumetric representation of a scene with a mesh object. The left image shows the user eye, screen, and mesh object. The right image shows the depth layer decomposition. This example has a depth complexity of four yielding four depth images. For each screen pixel there will be four intersection point/normal pairs that are stored (sorted in depth order). If there are less than four intersection points for a screen pixel (sx, sy), there will be a hard-coded null value for the remaining images.

3.3.1 Depth Peeling as an Implicit Representation of Surfaces

Now we adapt depth peeling to generate an implicit representation of a surface mesh. One can think of the output from depth peeling as a 2D grid of rays with intersection points along them where the intersection points have an associated color property. Our adaptation simply stores additional properties for these intersection points such as surface normals (see figure 3.1). We call the sequence of depth layers a *Depth Peel Image*, DPI. The key observation is that the DPI yields a compact decomposition of the volume of a scene proportional to the surface area of the mesh in the scene. Figure 3.1 shows a 2D scene with different depth layers. Note that for a single ray the depth layers partition the ray into inside and outside regions. Instead of discretizing the ray storing information at each grid point along the ray, we need only keep track of at most four points. In 3D this compact representation substantially improves the

runtimes of our editing, smoothing, and contouring algorithms from $O(n^3)$ to $O(n^2)$ (here n is the resolution of the screen).

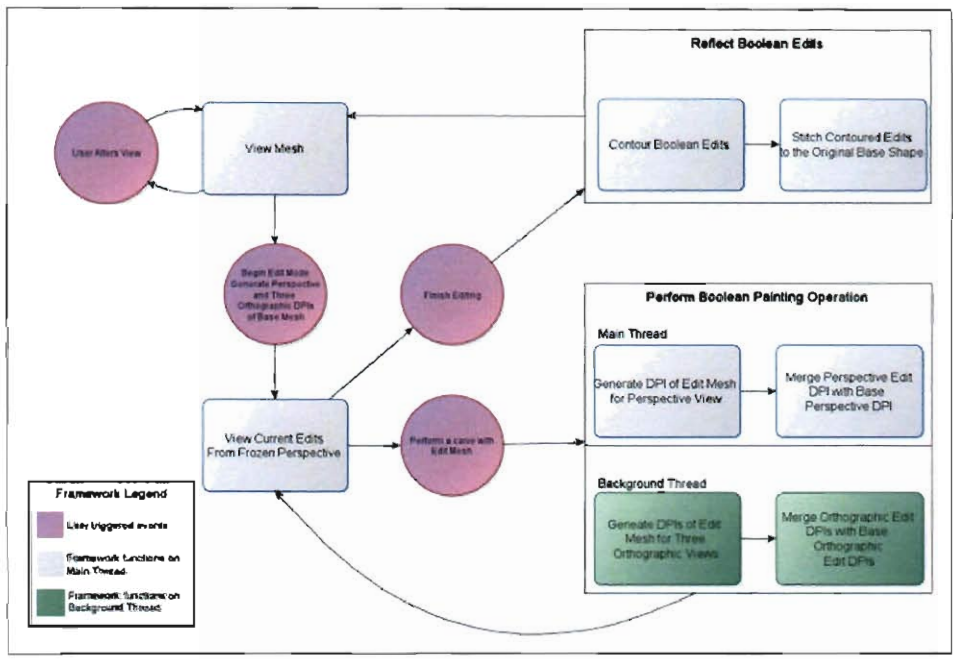


Figure 3.2 : Diagram showing the organization of computation within our framework.

Thus far we adapted depth peeling to generate an implicit representation of a surface mesh (DPI), but in practice we wish to output surface meshes from this implicit representation using a contouring algorithm. With our existing representation, we could generate a 3D uniform grid of hermite data with information along the grid edges parallel with the view direction by discretizing along the screen rays. From here we could create surfaces using a contouring approach from a suite of options: marching cubes, dual contouring, etc. This would not yield accurate results because we would not have sufficient information along grid edges orthogonal to the viewing direction. To account for this, we generate sequences of DPIs from three orthogonal

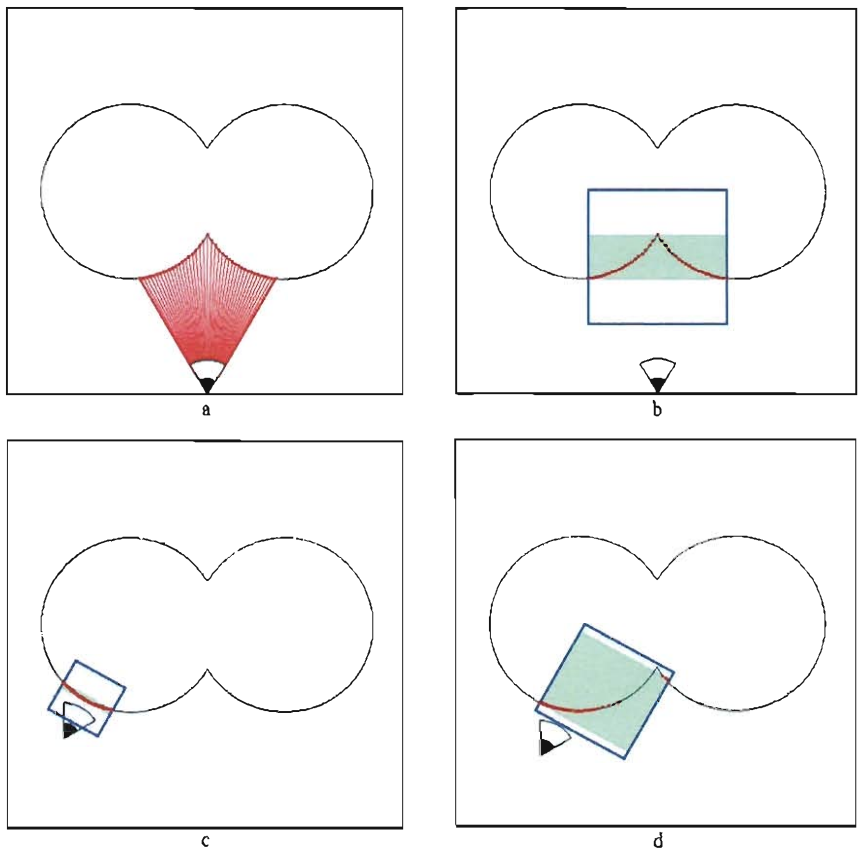


Figure 3.3 : Depicts 2D examples of our view-dependent algorithm for generating the bounds of the uniform grid from a user's view. The image in a) shows a user s view on a 2D shape. Rays are shot from the user's eye position into the scene and intersect with the base object (in 3D, the rays correspond to screen pixels). These intersection points are mapped into the user's coordinate space and a minimal bounding box is computed (the green region aligned with the user's perspective). After the algorithm concludes, we have a uniform cubical grid to perform painting operations. Images b-d show the generated bounding boxes from different perspectives.

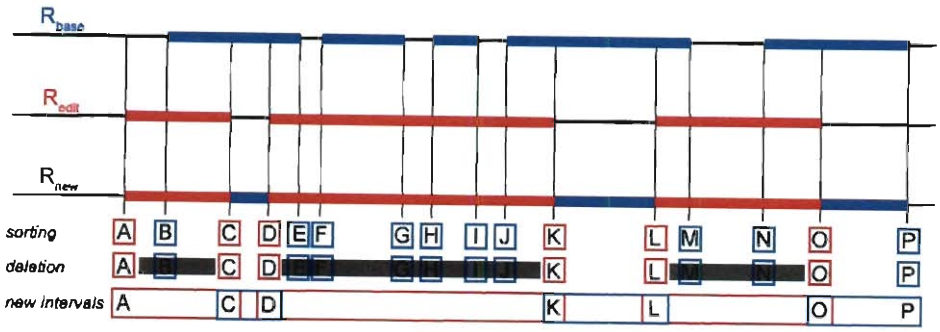


Figure 3.4 : Visual walk-through of our painting algorithm for a ray. The top lines show the intervals along R_{base}, R_{edit}, and R_{new}. The next lines show the ordered intersection points, followed by the eliminated points in the edit regions, and the final intervals.

will call our collection of four DPIs as View-dependent Depth Peel Image (VDPI).

3.4.1 Painting operations

In physical sculpting, artists use different materials to mold sculptures. Likewise, we wish to associate different materials with mesh operations. With this paradigm the primary editing operation is painting strokes of different materials, or colors, using surface meshes. Note in this workflow, subtraction operations are treated as painting with an "empty" material. Our VDPI representation is a 2D grid of 1D volumes with intersection information which allows us to paint a $VDPI_{base}$ with an $VDPI_{edit}$ on a ray-by-ray basis. As long as the rays from both sets overlap, we can perform volume painting operations on each corresponding ray pair. This can easily be guaranteed as we sample the surface meshes during the creation of the VDPIs since both sets have the same orientation and sampling frequency.

Our VDPI representation decomposes a ray into color regions. Painting a base ray, R_{base}, with an edit ray, R_{edit}, updates the base ray in the following fashion. Given

rays R_{base} and R_{edit} with corresponding sequences of hermite data H_{base} and H_{edit} our ray painting algorithm is as followed:

Figure 3.4.1 visually depicts our painting algorithm. Our ray painting technique performs three major steps: merging and sorting the hermite data points from the base and edit rays (line 1), removing hermite data points from the resulting ray that lie inside painted intervals along the edit ray (lines 3 - 10), and creating new intervals on the resulting ray (lines 13 - 19). We begin our algorithm by merging the two list of hermite data. During the merging we also sort the hermite data by increasing depth value. Since we wish to keep all of the intervals from R_{edit}, we remove all hermite data on the R_{new} that have depth values inside intervals on R_{edit}. After this deletion pass, we make new color intervals using the remaining hermite data. Subtle issues arise when intervals on R_{base} have endpoints at the same depth as an endpoint on R_{edit}, let us call them R_{base_i} and R_{edit_j}. In this case we ensure correctness of the algorithm by the placing R_{base_i} before R_{edit_i} if R_{base_i} is at the end of a region with a different color than that of R_{edit_i}. For all other cases, we place R_{base_i} after R_{edit_i}.

3.4.2 Smoothing operations

After sculptors mold their base shapes, they move to a refinement stage which includes adding fine details and/or smoothing portions of their models. Within our framework, we provide an intuitive approach for locally smoothing surfaces using our *View-dependent depth peel image*. There were a few implicit smoothing approaches that we could take. We first attempted a continuous convolution approach. We converted the rays in our VDPI to piecewise constant splines that had a value of 1 on side portions of a object and 0 otherwise (Note: for simplicity we smooth objects of a single material). Afterwards we implemented an algorithm to exactly reproduce the

Algorithm 3.2 Painting an existing 1D ray, R_{base}, with an edit ray, R_{edit}

Require: $\{R_{base_i}\}$, the intersection points for R_{base}

Require: $\{R_{edit_i}\}$, the intersection points for R_{edit}

Ensure: R_{new}, R_{base} altered by R_{edit}

1: $ipts = sort(R_{base}, R_{edit})$ // We sort the intersection points from lowest depth value to highest depth value. If R_{base_i} and R_{edit_j} have the same depth value, we place R_{base_i} before R_{edit_j} if R_{base_i} is at the end of a color region for R_{base}.

2: $pos = 0$

3: **for** $i = 0$ **to** $length(R_{base}) - 1$ step 2 **do**

4: **while** $pos < R_{base_{i+1}}$ **do**

5: **if** $R_{base_i} < ipts_{pos} < R_{base_{i+1}}$ **then**

6: $ipts = ipts - ipts_{pos}$

7: **end if**

8: $pos = pos + 1$

9: **end while**

10: **end for**

11: $R_{new} = \emptyset$

12: **if** $length(ipts) > 0$ **then**

13: $R_{new} = R_{new} \cup ipts_0$

14: **for** $i = 1$ **to** $length(ipts)$ **do**

15: **if** $color(ipts_i) \neq color(last(R_{new}))$ **then**

16: $R_{new} = R_{new} \cup ipts_i$ // The color of the copy of $ipts_i$ is $color(last(R_{new}))$

17: **end if**

18: $R_{new} = R_{new} \cup ipts_i$ // The color of the copy of $ipts_i$ is not changed

19: **end for**

20: **end if**

21: **return** R_{new}

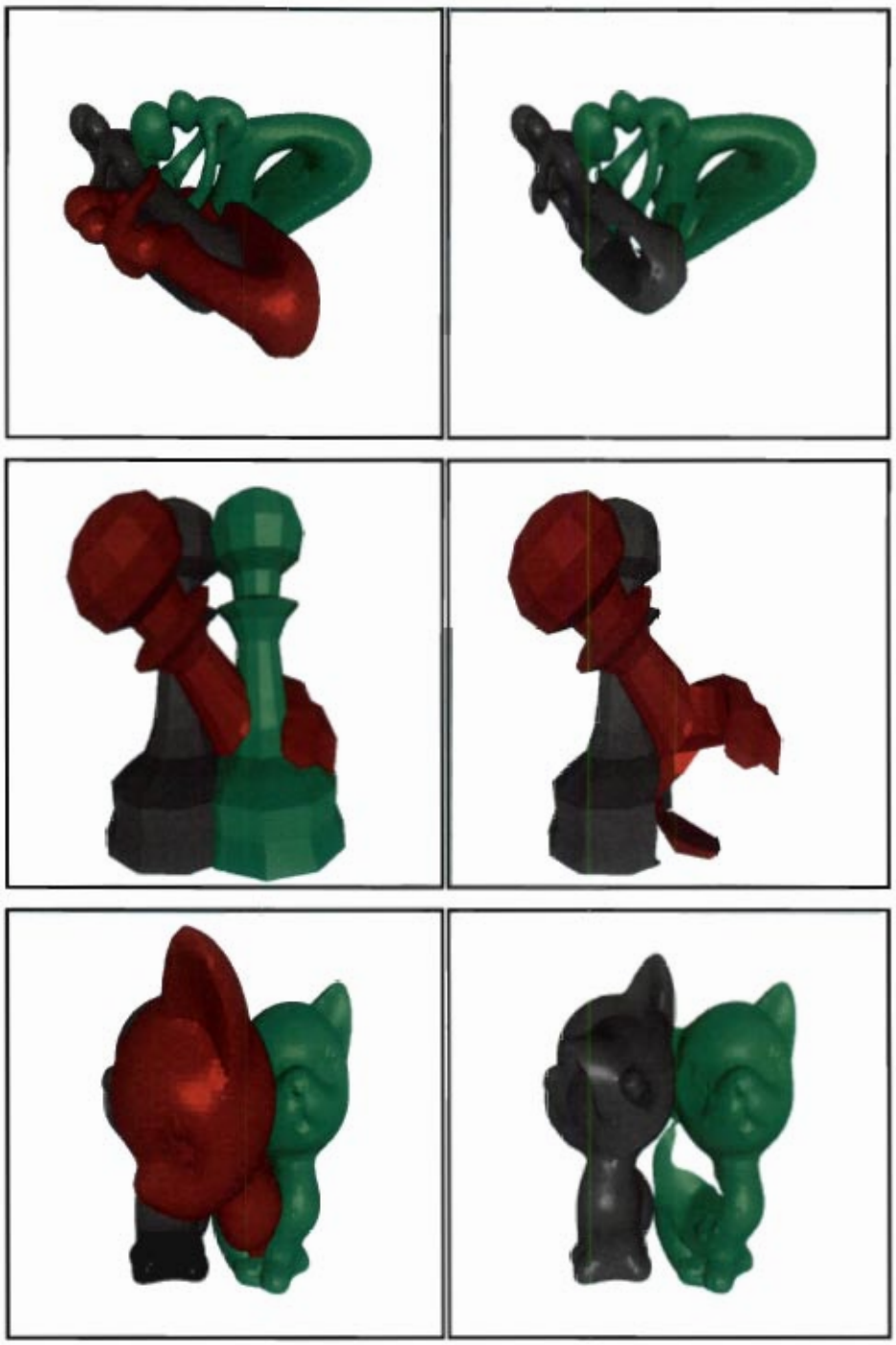


Figure 3.5 : Image sequences of painting operations using arbitrary closed meshes. Each of the left images contain three painting operations of a polygonal mesh. The right images show two of the three paintings for the corresponding left image. All of the painting operations were performed in less than a second. Convex shapes can perform in realtime; however, paintings with more complexity have larger generation times.

continuous convolution of the 2D hat function with the set of 1D piecewise constant splines. Finally we used the $\frac{1}{2}$ threshold and generated new rays with inside/outside intervals and normals given by the gradient of the convolved splines. This yielded discontinuities as surfaces crossed rays of the grid. Introducing new non-zero functions along the crossed ray added sharp bumps in the 2D convolution. In practice this meant we would get a different smoothing, depending the location of the surface with respect to our inferred volumetric grid.

In search of a technique that would be less sensitive to the location and orientation of the target surface we made the following observation. As a surface crosses into a grid cell, the percent of the cell inside the surface changes continuously. With this insight, we present an algorithm for performing local smoothing based on the percent of grid cells inside of our target surface. We begin by creating a *volume* grid by centering unit cubes around points in our uniform grid with a manhattan distance of one from the surface of our object to smooth. We compute and store the percent of the unit cube inside of the shape inside our new volume grid. Figure 3.6 shows a 2D example of a star. The upper-right image shows the intersection of the grid with the star and the wedge of partial volumes used in our volume computation. We also approximate gradients at these grid points with a Sobel-operator on the grid. Note that we do not perform computations on the entire grid, merely a wedge of voxels around the boundary of the target shape. We enforce the percent volume to be one for grid points external to the shape and the wedge of voxels and zero otherwise. This allows us not to generate a n^3 grid, but a hybrid grid of size $O(n^2)$. The $\frac{1}{2}$ contour from this new volume grid would yield a close approximation to the original shape similar to that returned from marching cubes. Afterwards, we perform Laplacian smoothing on the volume grid with a user-determined kernel size. With these smoothed volumes,

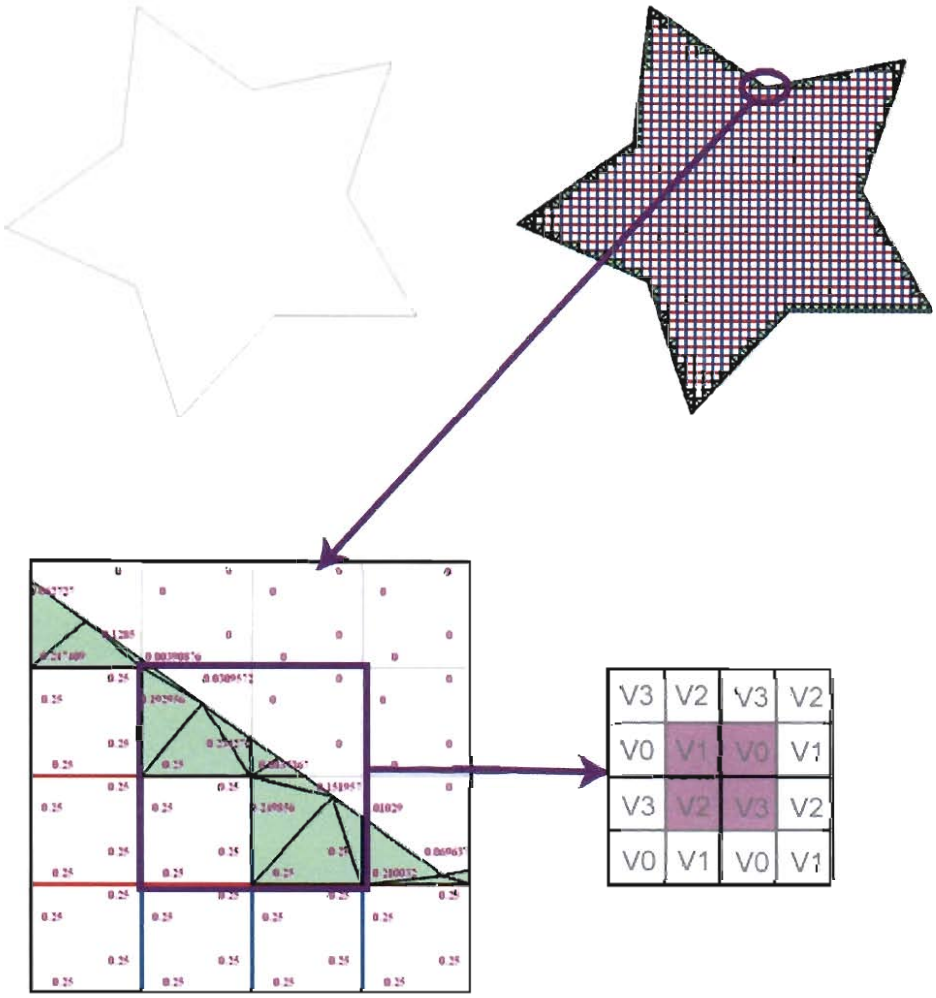


Figure 3.6 : Computing partial volumes. The star shape (upper-left) image is embedded in a grid (upper-right). We zoom in our the elliptical area on the bottom-left image. In the zoomed image, each cell has four values (V0-V3) where VI equals the percent of VI contained in the target shape. For each grid point in our wedge of voxels, we add the percents from the surrounding colored regions.

we create new hermite data along the rays at the $\frac{1}{2}$ threshold. We also approximate normals at these points with a Sobel operator applied locally to the grid. This results in a smoothed VDPI which we can perform subsequent operations on. To decrease the runtime of this algorithm, we do not perform calculations on the entire grid. We bound our calculations to grid points with distance from the surface of less than twice the user-determined kernel size. This improves the runtime of our algorithm from $O(n^3)$ to $O(n^2)$ where n is the resolution of the grid.

In practice users determine a region for smoothing (a rectangular prism). Inside this region the user would expect to have a fully smoothed surface. Outside of this region the user would expect for the surface to gradually reconcile itself with the original non-smoothed surface. We address this by increasing the region of operation by thrice the kernel size as opposed to twice (figure 3.7 highlights the additional region). Within this region, we linearly blend the original percent volume with the smoothed percent volume based on the distance from the boundary of the smoothing operation. So we now have new intersection information within the fully smooth and blend regions. We keep the original intersection information outside of these two regions. All information is retained in our VDPI for further processing.

3.5 Contouring Surface Edits

Up to this point, we have described how our framework represents surfaces in an implicit representation and how we use this representation in an editing workflow. Within the workflow, we have also shown you how we use this representation to sculpt new shapes using editing surface meshes and smooth surfaces. All of these features operate and are stored in our VDPI representation; however, we need to reflect these changes in a polygonal mesh. As a user changes views on the editing

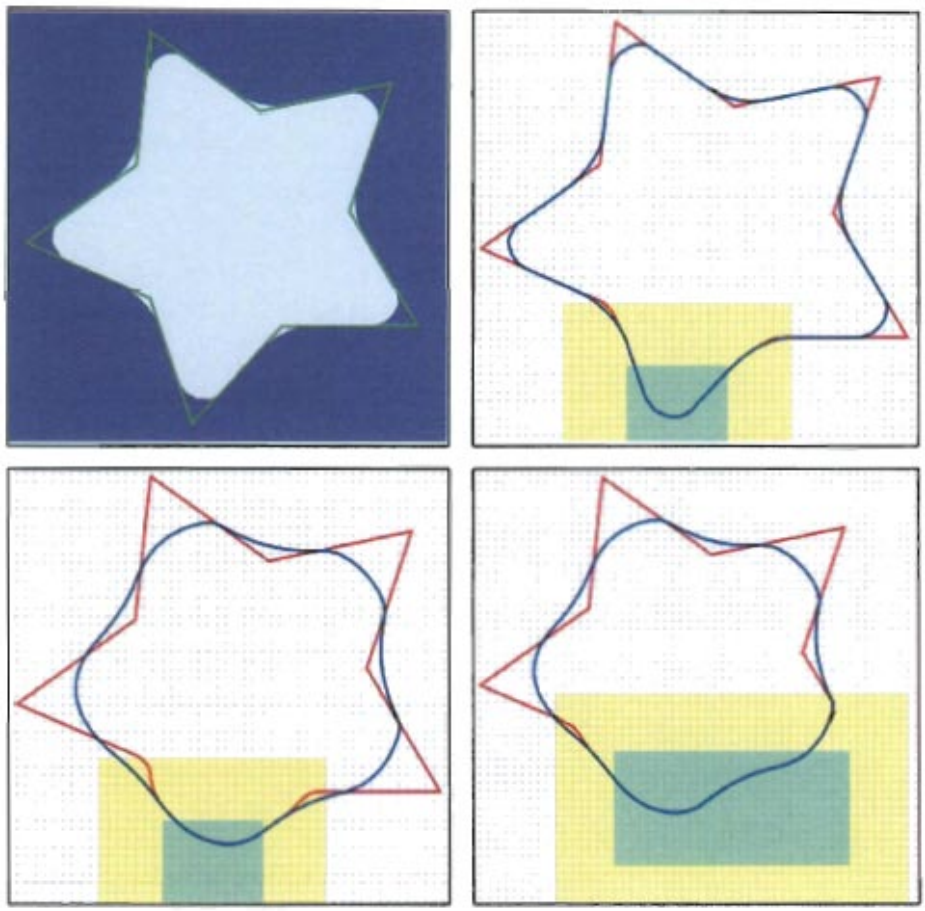


Figure 3.7 : 2D examples of smoothing regions of a star. The upper-left image shows a global smoothing of the star with the original star shape overlayed. The upper right image shows full smoothing region (cyan) and the blend region (yellow). This results in a star shape with a rounded spoke. The bottom two images use a larger kernel for smoothing which erodes substantial portions of the sharp features. We show results with different smoothing regions.

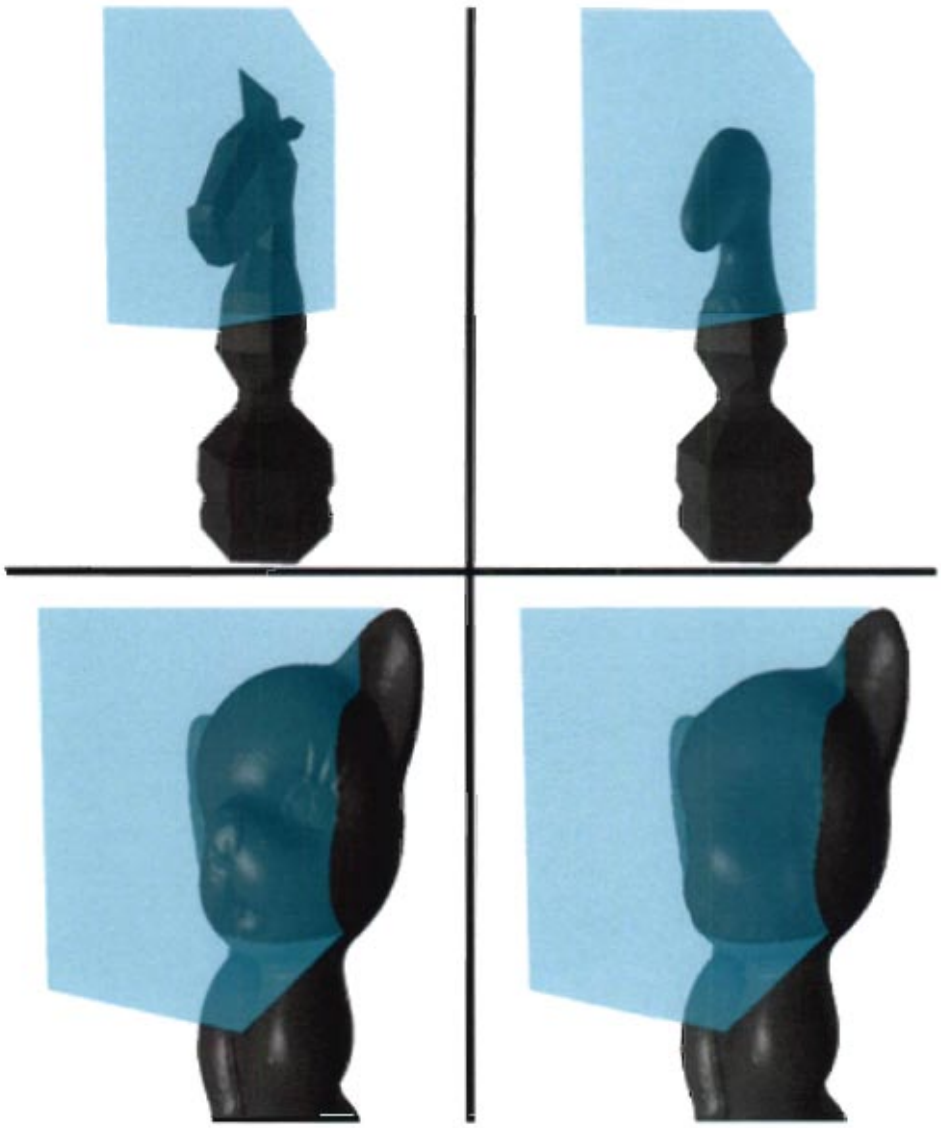


Figure 3.8 : 3D examples of smoothing surfaces with our algorithm.

Figure 3.9 : Image sequence that shows an overview of our stitching algorithm. The surface edits are bounded (yellow outline). The edits interior to the boundary are generated using Dual Contouring. The signs of the dual grid are determined by the polygons exterior to the bounded region for consistency. Then we stitch the original polygons external to the edit boundary with minimizers in the dual grid.

mesh, he/she may zoom in on a region of the altered surface to add intricate details. In this case, the new view would change the resolution of the inferred volumetric grid. So after a user is satisfied with a set of changes, we want to contour the surface edits and stitch these changes into the original base mesh.

3.5.1 Dual Contouring on VDPI

Since a VDPI gives hermite information on a uniform grid, we use Dual Contouring [14] to generate closed-surface meshes from our implicit structure. Since our VDR is proportional in size to $O(n^2)$ as opposed to $O(n^3)$ (once again n is the grid resolution), we can drastically improve the time spent contouring the surface. In our implementation, we keep a hashtable of Quadratic Error Functions (QEF refer to [14]). As we traverse the rays of the VDR we simply map the intersection data to the proper cell QEF via a hash on the grid position. Afterwards we make a pass that

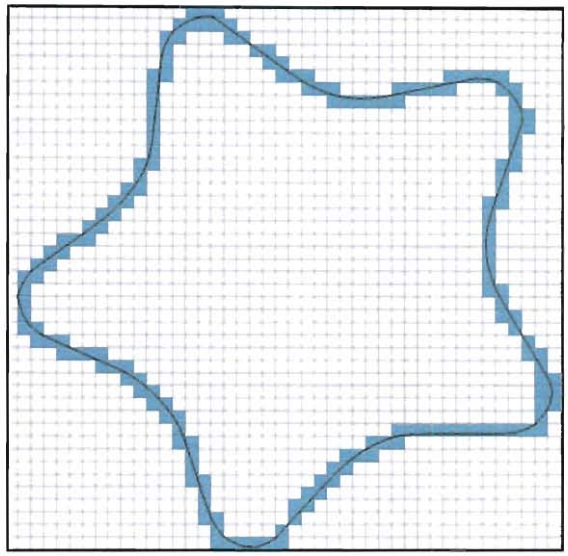


Figure 3.10 : Contouring example with the smoothed star. The cells that are used to generate the contour are highlighted. We use a relatively small portion of the grid cells.

generates the surface topology across dual grid edges with sign changes. Figure 3.10 shows a 2D example of the visited cells and generated QEF minimizers.

3.5.2 Stitching Algorithm

When patching user edits to our base surface, we want to keep as much of the original surface unchanged as possible. We enforce this by bounding the user edits by a sub-cube of the uniform grid. With this bound we proceed with the following simple algorithm with further explanation of each step (figure 3.9 depicts the stitching process on a 3D mesh). First we make a pass removing all portions of polygons on the original mesh that are interior to the edit bounds. In this pass if a polygon is completely inside the edit region, we remove it from the mesh. Afterwards we use a scan line conversion algorithm similar to Bresenham's approach [81] with edges from the trimmed original surface that lie on the edit bounds. This algorithm assigns

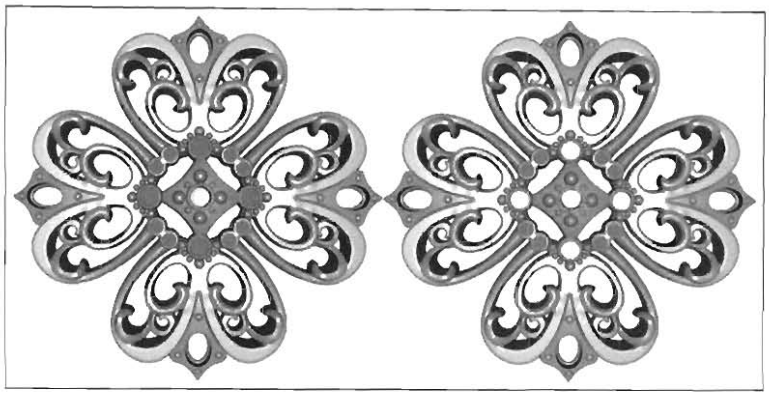


Figure 3.11 : A sequence of images showing contouring surface edits with different resolution grids. This image sequence shows the results of our fanning algorithm that stitches the exterior unaltered polygons with the contoured surface edits. Note that the two polygonal sets could vary greatly in sampling size.

material attributes to grid points of intersected grid edges. The material assignments from our scan line conversion algorithm form closed loops on the faces of the edit bounds that are consistent with the external surface. We employ a grid traversal along the edit bounds to flood material assignments for grid points using these closed loops as constraints to ensure consistent materials across the edit boundary. Furthermore the algorithm fans the QEF minimizers of intersected grid cells with the corresponding external polygons. In practice we triangulate trimmed polygons. This ensures that either one or two vertices of a trimmed polygon lie outside of the edit bounds. If there is only one external vertex, our fanning is trivial. If there are two, we connect scan converted minimizers to the closest external vertex. We then close the remaining triangle hole. We conclude the section with examples of reflecting surface edits with different grid resolutions. Figure 3.14 shows the fanning process on the edit bounds.

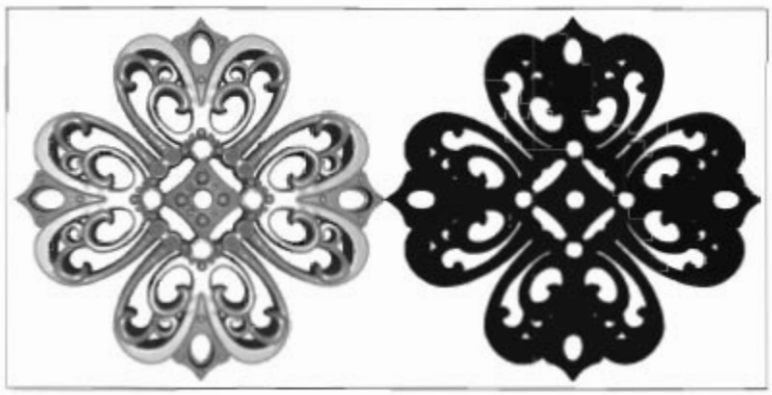


Figure 3.12 : Surface edits with an underlying $500 \times 500 \times 500$ grid.

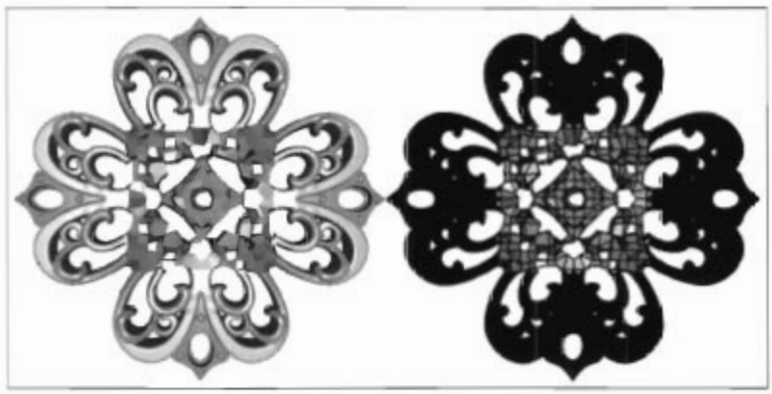


Figure 3.13 : Surface edits with an underlying $100 \times 100 \times 100$ grid.

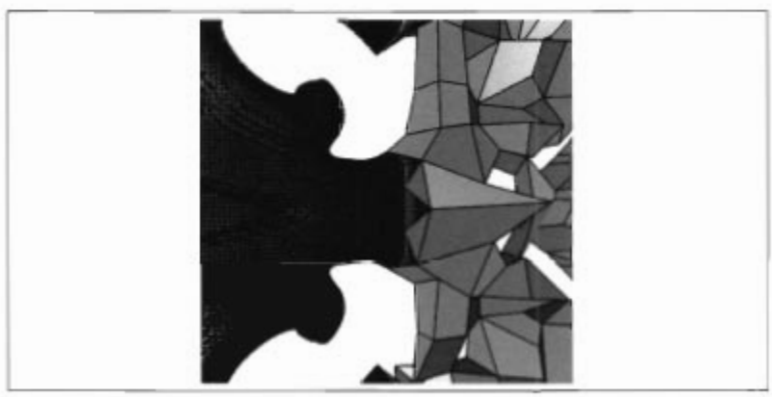


Figure 3.14 : Zoomed view of the $100 \times 100 \times 100$ grid example. The snowflake model is finely sampled and fans to minimizers inside our *Dual Contouring* representation.

3.6 Results

To test the performance of our methods, we developed a 3D surface editing application with functionality described in the previous sections. All results were taken on a 3GHz workstation. Performing painting operations using convex shapes achieves realtime surface editing performance (20-30fps). This performance is attributed to rendering only two layers for convex shapes. However, performing painting operations with arbitrary closed meshes yields sub-interactive performance (6-15fps) due to the information passing between the GPU and CPU during the acquisition of the depth peel images for an edit.

Our edit and contouring performance is substantially better than that of the most related work done by Chen et al. citelayerDepthNormal. We were not provided with an implementation of their work, however, we generated edit operations between surfaces (figure 3.5) with comparable triangle output and compared contouring runtimes. In practice, the primary runtimes we are concerned with are that of the editing and smoothing operations because a user will spend the majority of interaction time editing a surface. While our runtime for painting is fast for convex shapes, it is slow for complex shapes. Furthermore, our smoothing operations run on the order of seconds which we consider acceptable for smoothing a user-defined region.

We provide a set of tables and images that show the performance of our surface editing application.

3.7 Conclusions and Future work

We have provided a framework for view-dependent surface editing using an implicit approach. We have developed a hybrid implicit representation for surfaces that we

Model	VDPI Construction	VDPI Merging	Surface Contouring
Fertility Statues	0.14secs	0.015secs	0.6secs
Pawns	0.09secs	0.06secs	0.009secs
Kittens	0.13secs	0.011secs	0.51secs

Table 3.1 : Timing results for Figure 3.5. Results taken on a 512^3 grid.

Model	Smooth Region Size ($voxels^3$)	Kernel Size	Smoothing
Knight	48000	5	3.82secs
Kitten	48000	5	5.17secs

Table 3.2 : Timing results for Figure 3.8. Results taken on a 256^3 grid.

Model	Average Edit Time	Edit Contouring	Surface Patching
Figure 3.6	0.027secs	0.381secs	0.472secs
Figure 3.6	0.023secs	0.166secs	0.189secs
Figure 3.6	0.031secs	0.47secs	0.91secs

Table 3.3 : Timing results for carvings in figure 3.15. Results taken on a 512^3 grid.

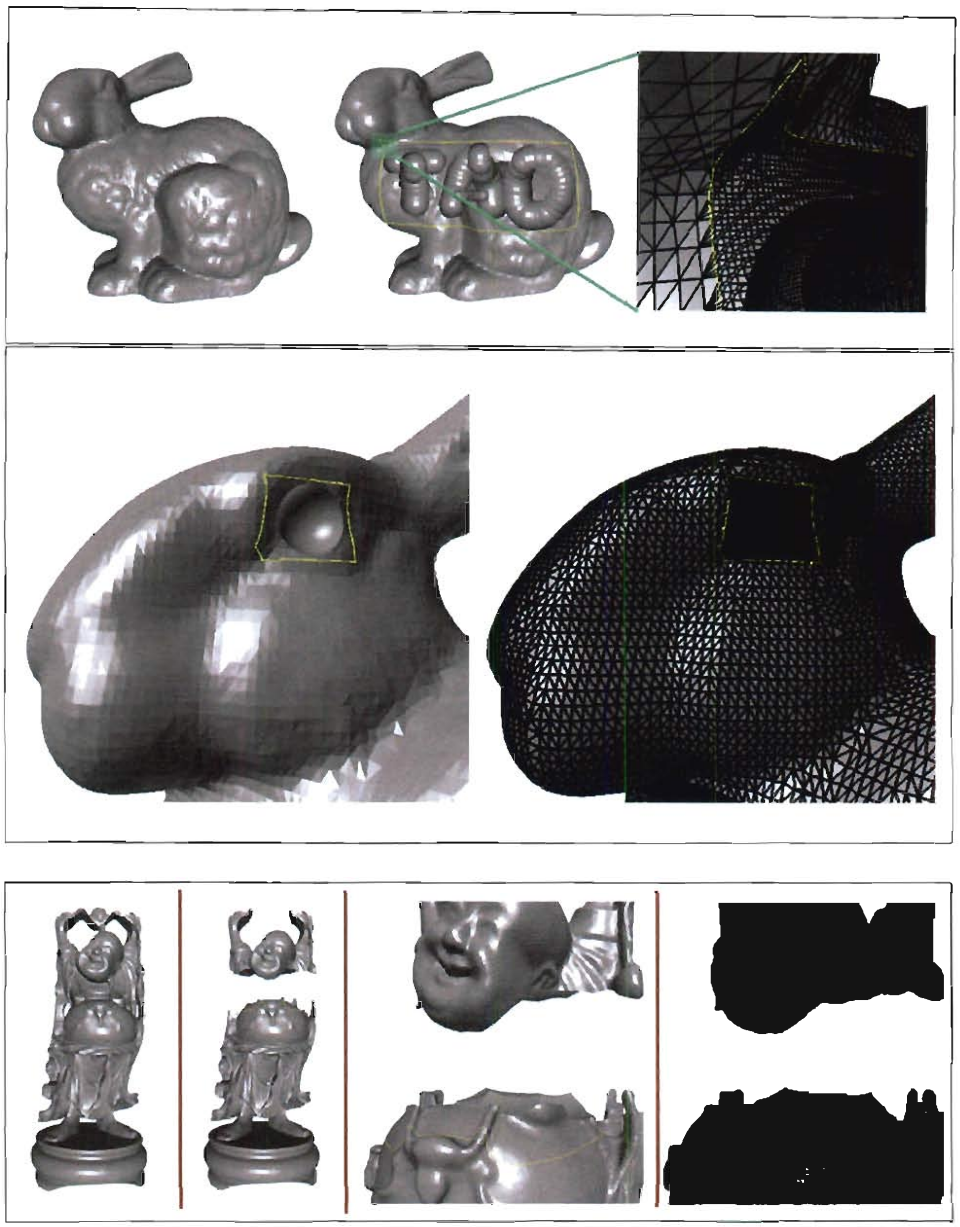


Figure 3.15 : Edited surfaces. All edits were done on an grid of size 512^3.

use to perform boolean and smoothing operations between surface meshes. Although we have a working application, the performance for editing operations is not desirable for complex shapes. The major bottleneck in our editing process is passing hermite data from the GPU to the host machine.

In the future, we would like to explore pushing all computations onto the GPU and producing a surface mesh at the end of all edits. This would alleviate the time needed to pass information from the GPU to the CPU multiple times during each edit. We would also like to remove the constraint of having a user freeze the camera position while editing. This involves viewing our implicit representation from oblique directions. There may be techniques from light field rendering that can be adapted to address this problem.

Chapter 4

Moving Least Squares Deformation

In the previous chapters we focused on viewing and editing volumetric information. In this chapter we shift to deforming volumetric images. Image deformation has a number of uses from animation, to morphing [82] and medical imaging [83]. To perform these deformations, the user selects some set of handles to control the deformation. These handles may take the form of points [84], lines [85], or even polygon grids [86]. As the user modifies the position and orientation of these handles. the image should deform in an intuitive fashion. We view this deformation as a function f that maps points in the undeformed image to the deformed image. Applying the function f to each point v in the undeformed image creates the deformed image. Now consider an image with a set of handles p that the user moves to new positions q. For f to be useful for deformations it$f$ must satisfy the following properties:

- *Interpolation*: The handles p should map directly to q under deformation. (i.e: $f(p_i) = q_i$).
- *Smoothness*: f should produce smooth deformations
- *Identity*: If the deformed handles q are the same as the p, then f should be the identity function. (i.e: $q_i = p_i \Longrightarrow f(v) = v$).

These properties are very similar to those used in scattered data interpolation. The first two properties simply state that the function f interpolates the scattered

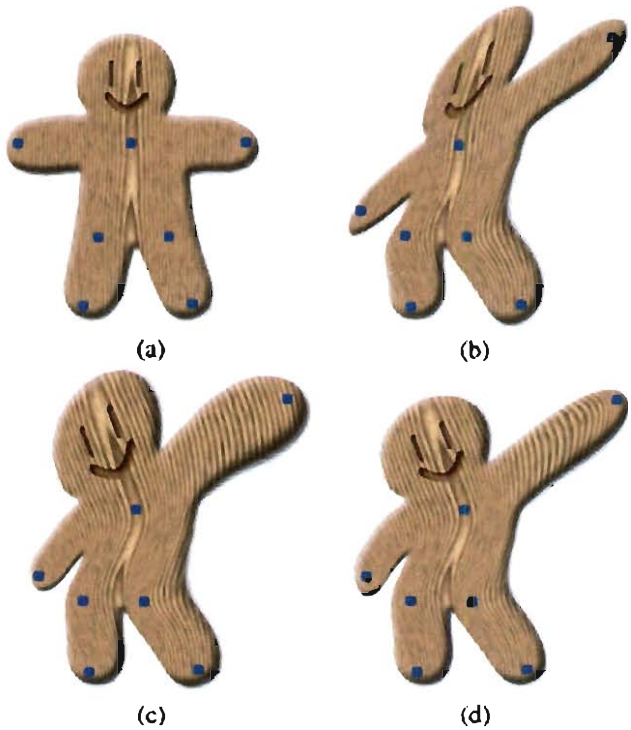


Figure 4.1 : Deformation using Moving Least Squares. Original image with control points shown in blue (a). Moving Least Squares deformations using affine transformations (b), similarity transformations (c) and rigid transformations (d).

data values and is smooth. The last property is sometimes referred to as linear precision in the approximation field. This property states that if data is sampled from a linear function, then the interpolant reproduces that linear function. Given these similarities, it comes as no surprise that many deformation methods borrow techniques from scattered data interpolation.

4.1 Related Work

Previous work on image deformation has focused on specifying deformations using different types of handles. Grid-based techniques such as free-form deformations [11, 87] parameterize the image using bivariate cubic splines to create C^2 deformations. Typically these methods require aligning grid lines corresponding to the control points of the spline with features of the image, which can be cumbersome for the user.

Beier et al. [85] improve upon these grid-based techniques and allow the user to specify the deformation using sets of lines. This method is based on Shepards interpolant [88] and creates smooth deformations. However, the authors note that their method produces complicated warps that can sometimes suffer from ghosts, undesirable folding in the deformation. Koba et al. [89] later generalized this technique to surface deformations.

Very few deformation methods investigate the type of transformations that are desirable for performing deformation. One notable exception is work based on thin-plate splines [84] that attempts to minimize the amount of bending in the deformation. Bookstein presents a deformation algorithm using the simplest deformation handle, a point, that uses radial basis functions with thin-plate splines. Figure 4.2 (left) shows an example of the deformation created with thin-plate splines for our example in figure 4.1. The deformation appears very similar to the affine-method in figure

4.1. In both cases, the test shape undergoes local non-uniform scaling and shearing, which is undesirable in many applications.

This chapter builds primarily on a recent paper by Igarashi et al. [90] that proposes a point-based image deformation technique for cartoon-like images in which the resulting deformations are as rigid-as-possible. Such deformation have the property that the amount of local scaling and shearing is minimized. (The concept of rigid-as-possible transformations was itself first introduced in Alexa [91].)

To produce rigid-as-possible deformations, Igarashi et al. triangulate the input image and solve a linear system of equations whose size is equal to the number of vertices in the triangulation. In contrast, our method creates deformations by solving a small linear system (2×2) at each point in a uniform grid (see Section 4 for details). Since we solve much smaller systems of equations, we can create very fast deformations of grids consisting of tens of thousands of vertices in real-time whereas Igarashi et al. report that their methods slows at 300 vertices on a 1 GHz machine. Due to the relatively small number of vertices, the deformations produced by Igarashi et al. may contain noticeable discontinuities as shown in figure 4.2. Figure 4.1 shows an equivalent deformation with our technique, which appears smooth.

4.2 Contributions

In this chapter, we propose an image deformation method based on linear Moving Least Squares. To construct deformations that minimize the amount of local scaling and shear, we restrict the classes of transformations used in Moving Least Squares to similarity and rigid-body transformations. By using MLS, we avoid the need to triangulate the input image (as done in Igarashi et al.) and produce deformations that are globally smooth.)

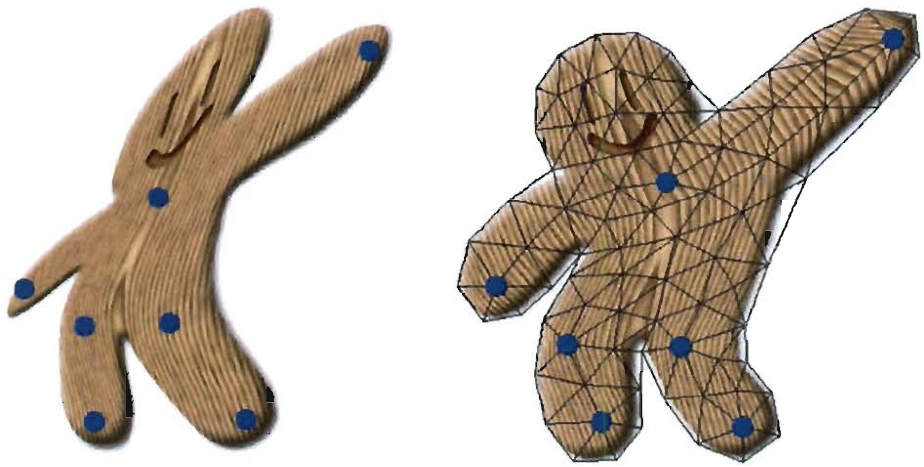


Figure 4.2 : Deformation of the test shape from figure 4.1 using thin-plate splines (left). The deformation is smooth but lacks realism. On the right we use the method by Igarashi et al. shown with triangulation (right). The lack of smoothness is clearly visible in the wood grain.

Next, we derive closed-form formulas for both similarity and rigid MLS deformations. These formulas are simple, easy to implement and provide real-time deformations. This derivation relies on a surprising and little-known relationship between similarity transformations and rigid transformations that minimize a common least squares problem. As opposed to Igarashi et al., our formulas do not require the use of a general linear solver. As a natural extension of our point-based method, we extend our MLS deformation method from sets of points to sets of line segments and again provide closed-form expressions for the resulting deformation method.

4.3 Moving Least Squares Deformation

Here we consider building image deformations based on collections of points with which the user controls the deformation. Let p be a set of control points and q the deformed positions of the control points p. We construct a deformation function

f satisfying the three properties outlined in the introduction using Moving Least Squares [92]. Given a point v in the image, we solve for the best affine transformation $l_v(x)$ that minimizes

$$\sum_i w_i |l_v(p_i) - q_i|^2 \tag{4.1}$$

where p_i and q_i are row vectors and the weights w_i have the form

$$w_i = \frac{1}{|p_i - v|^{2\alpha}}$$

Because the weights w_i in this least squares problem depend on the point of evaluation v, we call this a *Moving Least Squares* minimization. Therefore, we obtain a different transformation $l_v(x)$ for each v.

Now we define our deformation function f to be $f(v) = l_v(v)$. Observe that as v approaches p_i, w_i approaches infinity and the function f interpolates, (i.e; $f(p_i) = q_i$) Furthermore, if $q_i = p_i$, then each $l_v(x) = x$ for all x and, therefore, f is the identity transformation $f(v) = v$. Finally, this deformation function f has the property that it is smooth everywhere (except at the control points p_i when $\alpha \leq 1$).

Now since $l_v(x)$ is an affine transformation, $l_v(x)$ consists of two parts: a linear transformation matrix M and a translation T.

$$l_v(x) = xM + T \tag{4.2}$$

We can actually remove the translation T from this minimization problem further simplifying these equations. Equation 4.1 is quadratic in T. Since the minimizer is where the derivatives with respect to each of the free variables in $l_v(x)$ are zero, we can solve directly for T in terms of the matrix M. Taking the partial derivatives with

respect to the free variables in T produces a linear system of equations. Solving for T yields that

$$T = q_* - p_* M$$

where p_* and q_* are weighted centroids.

$$\begin{aligned} p_* &= \frac{\sum_i w_i p_i}{\sum_i w_i} \\ q_* &= \frac{\sum_i w_i q_i}{\sum_i w_i} \end{aligned}$$

With this observation we can substitute T into equation 4.2 and rewrite $l_v(x)$ in terms of the linear matrix M.

$$l_v(x) = (x - p_*)M + q_* \tag{4.3}$$

Based on this insight, the least squares problem of equation 4.1 can be rewritten as

$$\sum_i w_i |\hat{p}_i M - \hat{q}_i|^2 \tag{4.4}$$

where $\hat{p}_i = p_i - p_*$ and $\hat{q}_i = q_i - q_*$. Notice that Moving Least Squares is very general in that the matrix M does not have to be a fully affine transformation. In fact, this framework allows us to investigate different classes of transformation matrices M. In particular, we are interested in the case where M is a rigid transformation. However, we first examine the case where M is an affine transformation as the derivation is the simplest. Next we construct deformations with similarity transformations and show how these solutions can be used to find closed-form solutions to Moving Least Square deformations with rigid transformations.

4.3.1 Affine Deformations

Finding an affine deformation that minimizes equation 4.4 is straightforward using the classic normal equations solution.

$$M = \left(\sum_i \hat{p}_i^T w_i \hat{p}_i \right)^{-1} \sum_j w_j \hat{p}_j^T \hat{q}_j$$

Though this solution requires the inversion of a matrix, the matrix is a constant size (2×2) and is fast to invert. With this closed-form solution for M we can write a simple expression for the deformation function $f_a(v)$.

$$f_a(v) = (v - p_*) \left(\sum_i \hat{p}_i^T w_i \hat{p}_i \right)^{-1} \sum_j w_j \hat{p}_j^T \hat{q}_j + q_* \tag{4.5}$$

Applying this deformation function to each point in the image creates a new, deformed image.

While the user creates these deformations by manipulating the points q, the points p are fixed. Since the p do not change during deformation, much of equation 4.5 can be precomputed yielding very fast deformations. In particular, we can rewrite equation 4.5 in the form

$$f_a(v) \sum_j A_j \hat{q}_j + q_*$$

where A_j is a single scalar given by

$$A_j = (v - p_*) \left(\sum_i \hat{p}_i^T w_i \hat{p}_i \right)^{-1} \hat{p}_j^T$$

Notice that, given a point v, everything in A_j can be precomputed yielding a simple, weighted sum. Table [1] provides timing results for the examples in this

paper, which shows that these deformations may be performed over 500 times per second in our examples.

Figure 4.1 (b) illustrates this affine Moving Least Squares deformation applied to our test image. Unfortunately, the deformation does not appear very desirable due to the stretching in the arms and torso. These artifacts are created because affine transformations include deformations such as non-uniform scaling and shear. To eliminate these undesirable deformations, we need to consider restricting the linear transformation $l_v(x)$. In particular, we modify the class of deformations $l_v(x)$ by restricting the transformation matrix M to similarity and rigid-body transformations.

4.3.2 Similarity Deformations

While affine transformations include effects such as non-uniform scaling and shear, many objects in reality do not undergo even these simple transformations. Similarity transformations are a special subset of affine transformations that only include translation, rotation, and uniform scaling.

To alter our deformation technique to only use similarity transformations, we constrain the matrix M to have the property that $M^T M = \lambda^2 I$ for some scalar λ. If M is a block matrix of the form

$$M = \begin{pmatrix} M_1 & M_2 \end{pmatrix}$$

where M_1, M_2 are column vectors of length 2, then restricting M to be a similarity transform requires that $M_1^T M_1 = M_2^T M_2 = \lambda^2$ and $M_1^T M_2 = 0$. This constraint implies that $M_2 = M_1^{\perp}$ such that $(x, y)^{\perp} = (-y, x)$. Though restricted, the minimization problem from equation 4.4 is still quadratic in M_1 and can be rephrased as finding the column vector M_1 that minimizes

$$\sum_i w_i \left| \begin{pmatrix} \hat{p}_i \\ -\hat{p}_i^{\perp} \end{pmatrix} M_1 - \hat{q}_i^T \right|^2$$

This quadratic function has a unique minimizer, which yields the optimal transformation matrix M

$$M = \frac{1}{\mu_s} \sum_i \begin{pmatrix} \hat{p}_i \\ -\hat{p}_i^{\perp} \end{pmatrix} \begin{pmatrix} \hat{q}_i^T & -\hat{q}_i^{\perp T} \end{pmatrix} \tag{4.6}$$

where

$$\mu_s = \sum_i w_i \hat{p}_i \hat{p}_i^T$$

Similar to the affine deformations, the user manipulates the q to produce the deformation while the p remain fixed. Using this observation, we write the deformation function $f_s(v)$ in a form that allows us to precompute as much information as possible. $f_s(v)$ is then

$$f_s(v) = \sum_i \hat{q}_i \left(\frac{1}{\mu_s} A_i \right) + q_*$$

where μ_s and A_i depend only on the p_i, v and can be precomputed and A_i is

$$A_i = w_i \begin{pmatrix} \hat{p}_i \\ -\hat{p}_i^{\perp} \end{pmatrix} \begin{pmatrix} v - p_* \\ -(v - p_*)^{\perp} \end{pmatrix}^T \tag{4.7}$$

As expected, similarity MLS deformations preserves angles in the original image better than affine MLS deformations. (Transformations that strictly preserve angle are called conformal transformations and have been studied extensively in [93].) While approximate (or exact) angle preservation is a desirable property in many cases, allowing local scaling can often lead to undesirable deformations. Figure 4.1 (c) shows an example of applying the similarity Moving Least Squares deformation to our test

image. The result is a much more realistic looking deformation than (b). However, this deformation scales the size of the upper arm as it is stretched. To remove this scaling, we consider building deformations using only rigid transformations.

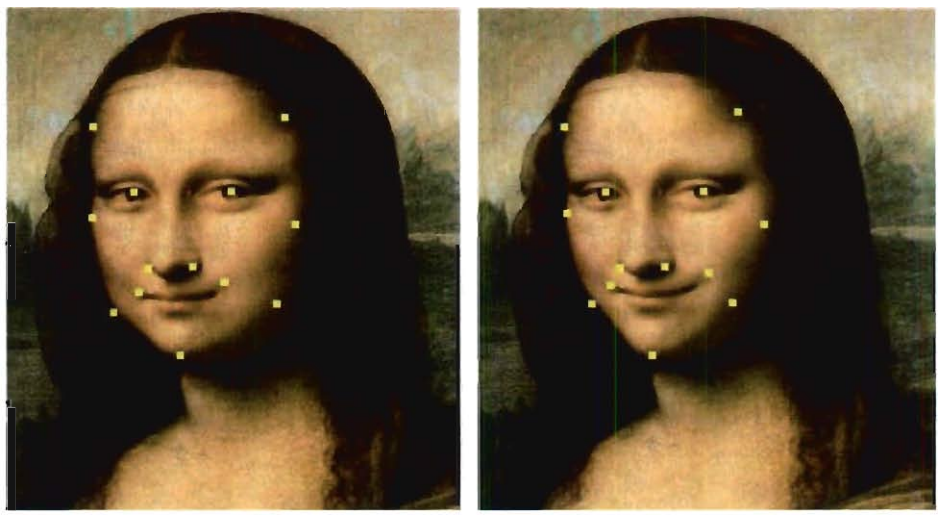


Figure 4.3 : Original image (left) and its deformation using the rigid MLS method (right). After deformation, the face is thinner and she is smiling.

4.3.3 Rigid Deformations

Recently, several works [91, 90] have shown that, for realistic shapes, deformations should be as rigid as possible; that is, the space of deformations should not even include uniform scaling. Traditionally researchers in deformation have been reluctant to approach this problem directly due to the non-linear constraint that $M^T M = I$. However, we note that closed-form solutions to this problem are known from the Iterated Closest Point community [94]. Horn shows that the optimal rigid transformation can be found in terms of eigenvalues and eigenvectors of a covariance matrix involving the points p_i and q_i. We show that these rigid deformations are related to the similarity deformations from section 4.3.2 via the following theorem.

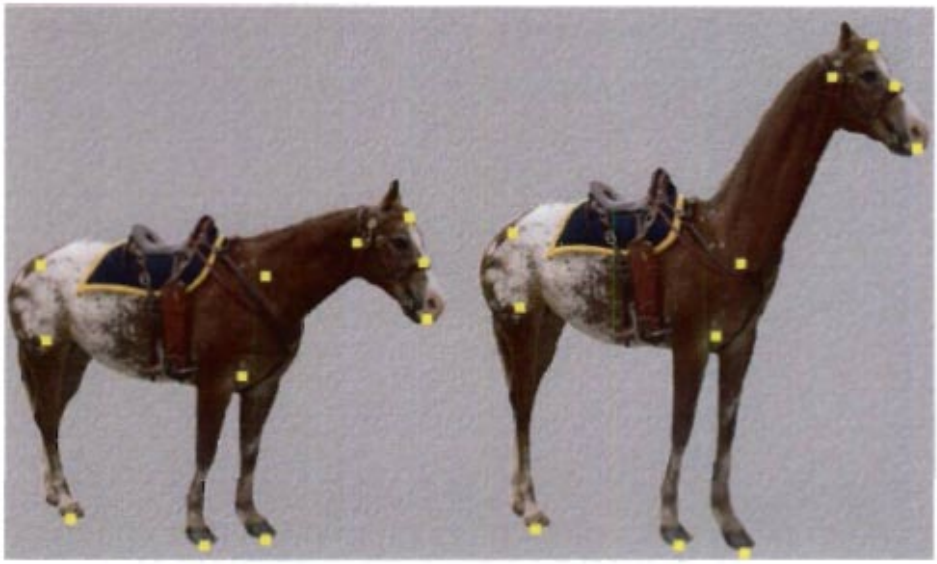


Figure 4.4 : Original image (left) and its deformation using the rigid MLS method (right).

Theorem 4.1

Let C be the matrix that minimizes the following similarity functional

$$\min_{M^T M = \lambda^2 I} \sum_i w_i \left| \hat{p}_i M - \hat{q}_i \right|^2$$

If C is written in the form λR where R is a rotation matrix and λ is a scalar, the rotation matrix R minimizes the rigid functional

$$\min_{M^T M = I} \sum_i w_i \left| \hat{p}_i M - \hat{q}_i \right|^2$$

Proof 4.1 First, we expand both of the above error functions into their quadratic forms yielding

$$\min_{R^T R=I,\lambda} \sum_i w_i \left(\lambda^2 \hat{p}_i \hat{p}_i^T - 2\lambda \hat{p}_i R \hat{q}_i^T + \hat{q}_i \hat{q}_i^T\right)$$
$$\min_{R^T R=I,\lambda} \sum_i w_i \left(\lambda^2 \hat{p}_i \hat{p}_i^T - 2\lambda \hat{p}_i M \hat{q}_i^T + \hat{q}_i \hat{q}_i^T\right)$$

These minimization problems are very similar. We find the matrices that minimize these error functions by differentiating the functions with respect to the free variables θ_j in R.

$$\sum_i w_i \left(-2\lambda \hat{p}_i \frac{\partial R}{\partial \theta_j} \hat{q}_i^T\right) = 0$$
$$\sum_i w_i \left(-2\hat{p}_i \frac{\partial R}{\partial \theta_j} \hat{q}_i^T\right) = 0$$

Now, unless $\lambda = 0$, which implies a degenerate transformation, these equations are equal. Since $C = \lambda R$, this implies that $\pm R$ minimizes the quadratic function using rigid transformations. The negative solution corresponds to a maximum while the positive solution is the minimum. QED

This theorem is valid in arbitrary dimension, however, it is very easy to apply in $2D$. Using this theorem, we find that the rigid transformation is exactly the same as equation 4.6 except that we use a different constant μ_r in the solution so that $M^T M = I$ is given by

$$\mu_r = \sqrt{\left(\sum_i w_i \hat{q}_i \hat{p}_i^T\right)^2 + \left(\sum_i w_i \hat{q}_i \hat{p}_i^{\perp T}\right)^2}$$

Unlike the similarity deformation $f_s(v)$, we cannot precompute as much information for the rigid deformation function $f_r(v)$. However, the deformation process can still be made very efficient. Let

$$\vec{f_r}(v) = \sum_i \hat{q}_i A_i$$

where A_i is defined in equation 4.7, which may be precomputed. This vector $\vec{f_r}(v)$ is a rotated and scaled version of the vector $v - p_*$. To compute $f_r(v)$, we normalize $\vec{f_r}$, scale by the length of $v - p_*$ (which also can be precomputed), and translate by q_*.

$$f_r(v) = |v - p_*| \frac{\vec{f_r}(v)}{|\vec{f_r}(v)|} + q_* \tag{4.8}$$

This method is slower than the similarity deformation due to the normalization; however, these deformations are still very fast as shown in table [1].

Figure 4.1 (d) shows this rigid deformation applied to the test image in (a). As opposed to the other methods, this deformation is quite realistic and almost feels as if the user is manipulating a real object. Figures 4.3 and 4.4 show additional examples of this rigid deformation method. In the figure with the Mona Lisa, we deform the image to create a thinner facial profile and make her smile. In the figure with the horse, we stretch the horses legs and neck to create a giraffe. Due to the use of rigid transformations, the deformation maintains rigidity and scale locally so that the body and head of the horse retain their relative shape.

4.4 Deformation with Line Segments

So far we have considered creating deformations with Moving Least Squares using only sets of points to control the deformation. In applications where precise control over curves such as profiles in the image is needed, points may be insufficient for specifying these deformations. One solution that allows the user to control curves

precisely is to convert these curves to dense sets of points and apply a point-based deformation [95]. The disadvantage of this approach is that the computation time of the deformation is proportional to the number of control points used and creating large numbers of control points adversely affects performance.

Alternatively, we desire a generalization of these Moving Least Squares deformations from section 4.3 to arbitrary curves in the plane. First, assume $p_i(t)$ is the i^{th} control curve and $q_i(t)$ is the deformed curve corresponding to $p_i(t)$. We generalize the quadratic function in equation 4.1 by integrating over each control curve $p_i(t)$ where we assume $t \in [0, 1]$.

$$\sum_i \int_0^1 w_i(t) |p_i(t)M + T - q_i(t)|^2 \, dt \tag{4.9}$$

where $w_i(t)$ is

$$w_i(t) = \frac{|p_i'(t)|}{|p_i(t) - v|^{2\alpha}}$$

and $p_i'(t)$ is the derivative of $p_i(t)$. (This factor of $|p'(t)|$ makes the integrals independent of the parametrization of the curve $p_i(t)$.) Now notice that, despite the integral, equation 4.9 is still quadratic in T and can be solved for in terms of the matrix M.

$$T = q_* - p_* M$$

where p_* and q_* are again weighted centroids.

$$p_* = \frac{\sum_i \int_0^1 w_i(t) p_i(t)\, dt}{\sum_i \int_0^1 w_i(t)\, dt} \tag{4.10}$$

$$q_* = \frac{\sum_i \int_0^1 w_i(t) q_i(t)\, dt}{\sum_i \int_0^1 w_i(t)\, dt} \tag{4.11}$$

Therefore, we rewrite equation 4.9 only in terms of M as

$$\sum_i \int_0^1 w_i(t) |\hat{p}_i(t) M - \hat{q}_i(t)|^2\, dt \tag{4.12}$$

where

$$\hat{p}_i(t) = p_i(t) - p_*$$

$$\hat{q}_i(t) = q_i(t) - q_*$$

Until now, $p_i(t)$ and $q_i(t)$ have been arbitrary curves. However, the integrals in equation 4.12 may be difficult to evaluate for arbitrary functions. Instead, we restrict these functions to be line segments and derive closed-form solutions for the deformations in terms of the end-points of these segments. Similar to section 4.3, we first consider affine transformations due to their relatively simple derivation and then move to similarity transformations, which we use to create closed-form solutions to the equivalent problem using rigid-body transformations.

4.4.1 Affine Lines

Since $\hat{p}_i(t)$, $\hat{q}_i(t)$ are line segments, we can represent these curves as matrix products

$$\hat{p}_i(t) = \begin{pmatrix} 1-t & t \end{pmatrix} \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix}$$

$$\hat{q}_i(t) = \begin{pmatrix} 1-t & t \end{pmatrix} \begin{pmatrix} \hat{c}_i \\ \hat{d}_i \end{pmatrix}$$

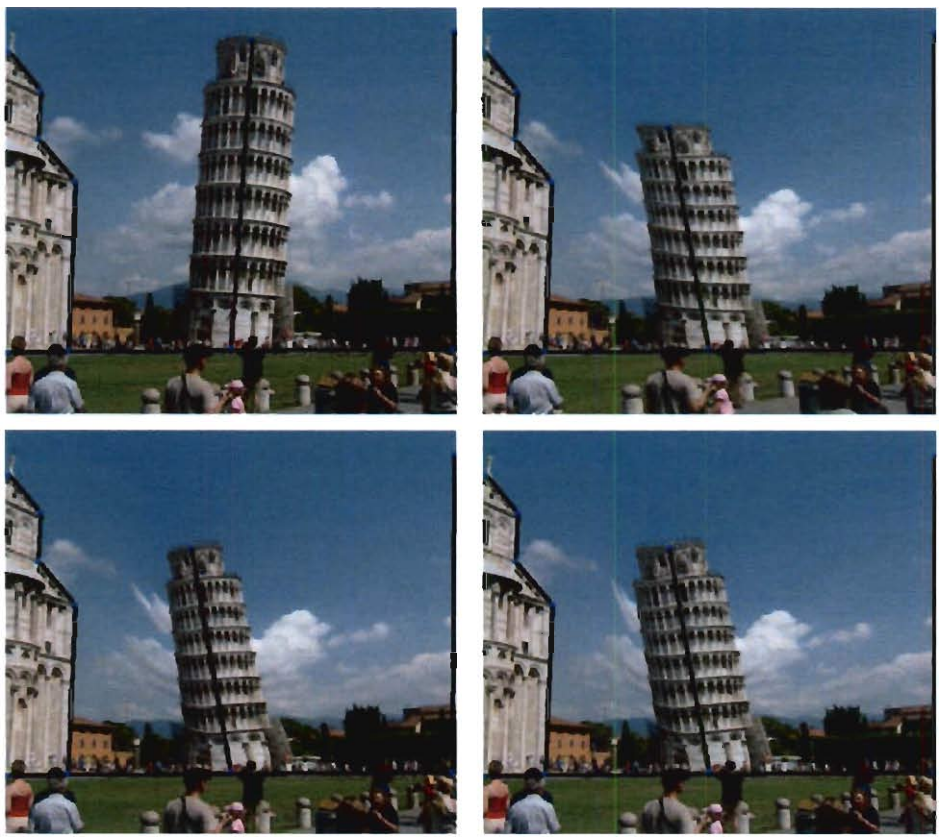


Figure 4.5 : Deformation of the Leaning Tower of Pisa. From left to right: original image, Affine MLS, Similarity MLS and Rigid MLS deformations.

where $\hat{a}_i$, $\hat{b}_i$ are the end-points of $\hat{p}_i(t)$ and $\hat{c}_i$, $\hat{d}_i$ are the end-points of $\hat{q}_i(t)$. Equation 4.12 is then written as

$$\sum_i \int_0^1 \left| (\ 1-t\ t\ ) \left(\begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix} M - \begin{pmatrix} \hat{c}_i \\ \hat{d}_i \end{pmatrix} \right) \right|^2 \tag{4.13}$$

whose minimizer is

$$M = \left(\sum_i \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix}^T W_i \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix} \right)^{-1} \sum_j \begin{pmatrix} \hat{a}_j \\ \hat{b}_j \end{pmatrix}^T W_j \begin{pmatrix} \hat{c}_j \\ \hat{d}_j \end{pmatrix}$$

where W_i is a weight matrix given by

$$W_i = \begin{pmatrix} \delta_i^{00} & \delta_i^{01} \\ \delta_i^{01} & \delta_i^{11} \end{pmatrix}$$

and the δ_i are integrals of the weight function $w_i(t)$ multiplied by the different quadratic polynomials.

$$\begin{aligned} \delta_i^{00} &= \int_0^1 w_i(t)(1-t)^2\,dt \\ \delta_i^{01} &= \int_0^1 w_i(t)t(1-t)\,dt \\ \delta_i^{11} &= \int_0^1 w_i(t)t^2\,dt \end{aligned}$$

These integrals have closed-form solutions for various values of α as follows (α is the exponent in the denominator of our weight function $w_i(t)$. Let a_i, b_i be the end-points of the line segment described by $p_i(t)$ and let

$$
\begin{aligned}
\Delta_i &= (a_i - v)^{\perp}(a_i - b_i)^T \\
\theta_i &= tan^{-1}\left(\frac{(b_i - v)(b_i - a_i)^T}{(b_i - v)^{\perp}(b_i - a_i)^T}\right) - tan^{-1}\left(\frac{(a_i - v)(a_i - b_i)^T}{(a_i - v)^{\perp}(a_i - b_i)^T}\right) \\
\beta_i^{00} &= (a_i - v)^{\perp}(a_i - v)^T \\
\beta_i^{01} &= (a_i - v)^{\perp}(v - b_i)^T \\
\beta_i^{11} &= (v - b_i)^{\perp}(v - b_i)^T
\end{aligned}
$$

The integrals then have the closed-form solution

$$
\begin{aligned}
\int_0^1 w_i(t)(1-t)^2\,dt &= \frac{|a_i - b_i|}{2\Delta_i^2}\left(\frac{\beta_i^{01}}{\beta_i^{00}} - \frac{\beta_i^{11}\theta_i}{\Delta_i}\right) \\
\int_0^1 w_i(t)t(1-t)\,dt &= \frac{|a_i - b_i|}{2\Delta_i^2}\left(1 - \frac{\beta_i^{01}\theta_i}{\Delta_i}\right) \\
\int_0^1 w_i(t)t^2\,dt &= \frac{|a_i - b_i|}{2\Delta_i^2}\left(\frac{\beta_i^{01}}{\beta_i^{11}} - \frac{\beta_i^{00}\theta_i}{\Delta_i}\right)
\end{aligned}
$$

When v is on the line segment defined by a_i and b_i, these integrals do not need to be evaluated because the function $f(v)$ interpolates the line segments. However, if v is on the extension of one of these line segments, $\Delta_i = 0$ and these integrals reduce to

$$
\begin{aligned}
\int_0^1 w_i(t)(1-t)^2\,dt &= \frac{|a_i - b_i|^5}{3((v - b_i)(b_i - a_i)^T)((a_i - v)(b_i - a_i)^T)^3} \\
\int_0^1 w_i(t)t(1-t)\,dt &= \frac{-|a_i - b_i|^5}{6((v - b_i)(b_i - a_i)^T)((a_i - v)(b_i - a_i)^T)^2} \\
\int_0^1 w_i(t)t^2\,dt &= \frac{|a_i - b_i|^5}{3((v - b_i)(b_i - a_i)^T)^3((a_i - v)(b_i - a_i)^T)}
\end{aligned}
$$

Note that these integrals can also be used to evaluate p_* and q_* from equation 4.10.

$$
\begin{aligned}
p_* &= \frac{\sum_i a_i(\delta_i^{00} + \delta_i^{01}) + b_i(\delta_i^{01} + \delta_i^{11})}{\sum_i \delta_i^{00} + 2\delta_i^{01} + \delta_i^{11}} \\
q_* &= \frac{\sum_i c_i(\delta_i^{00} + \delta_i^{01}) + d_i(\delta_i^{01} + \delta_i^{11})}{\sum_i \delta_i^{00} + 2\delta_i^{01} + \delta_i^{11}}
\end{aligned}
$$

As before, we write the deformation function $f_a(v)$ as

$$f_a(v) = \sum_j A_j \begin{pmatrix} \hat{c}_j \\ \hat{d}_j \end{pmatrix} + q_*$$

where A_j is a 1×2 matrix of the form

$$A_j = (v - p_*) \left(\sum_i \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix}^T W_i \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix} \right)^{-1} \begin{pmatrix} \hat{a}_i \\ \hat{b}_i \end{pmatrix}^T W_j$$

During the deformation, the end-points a_i and b_i of the line segment $p_i(t)$ are fixed while the user manipulates the end-points c_i and d_i of the line segments $q_i(t)$. Since A_j is independent of c_i and d_i, A_j can be precomputed.

Figure 4.5 shows an example deformation performed with line segments where we modify the Leaning Tower of Pisa to lean in the opposite direction and shrink the tower. The Affine MLS deformation shears the tower to the side instead of rotating the tower and does not appear to be realistic. To remove this shear effect, we restrict the matrix in equation 4.12 to be a similarity or rigid-body transformation.

4.4.2 Similarity Lines

Restricting equation 4.13 to similarity transforms requires that $M^T M = \lambda^2 I$ for some scalar λ. As noted in section 4.3.2, M can be parameterized using a single column vector M_1 yielding

$$\sum_i \int_0^1 \left| \begin{pmatrix} 1-t & 0 & t & 0 \\ 0 & 1-t & 0 & t \end{pmatrix} \left(\begin{pmatrix} \hat{a}_i \\ -\hat{a}^\perp \\ \hat{b}_i \\ -\hat{b}_i^\perp \end{pmatrix} M_1 - \begin{pmatrix} \hat{c}_j^T \\ \hat{d}_i^T \end{pmatrix} \right) \right|^2$$

This error function is quadratic in M_1. To find the minimizer, we differentiate with respect to the free variables in M_1 and solve the linear system of equations to obtain the matrix M.

$$M = \frac{1}{\mu_s} \sum_j \begin{pmatrix} \hat{a}_j \\ -\hat{a}_j^\perp \\ \hat{b}_j \\ -\hat{b}_j^\perp \end{pmatrix}^T W_j \begin{pmatrix} \hat{c}_j^T & \hat{c}_j^{\perp T} \\ \hat{d}_j^T & \hat{d}_j^{\perp T} \end{pmatrix} \tag{4.14}$$

where W_j is a weight matrix

$$W_j = \begin{pmatrix} \partial_j^{00} & 0 & \partial_j^{01} & 0 \\ 0 & \partial_j^{00} & 0 & \partial_j^{01} \\ \partial_j^{01} & 0 & \partial_j^{11} & 0 \\ 0 & \partial_j^{01} & 0 & \partial_j^{11} \end{pmatrix} \tag{4.15}$$

and μ_s is again a scaling constant, which has the form

$$\mu_s = \sum_i \hat{a}_i \hat{a}_i^T \partial_i^{00} + 2\hat{a}_i \hat{b}_i^T \partial_i^{01} + \hat{b}_i \hat{b}_i^T \partial_i^{11}$$

This deformation function has a very similar structure to the point-based similarity deformation. Using this matrix, we write $f_s(v)$ explicitly as

$$f_s(v) = \sum_j \begin{pmatrix} \hat{c}_j & \hat{d}_j \end{pmatrix} \left(\frac{1}{\mu_s} A_j\right) + q_*$$

where A_j is a 4×2 matrix.

$$A_j = W_j \begin{pmatrix} \hat{a}_j \\ -\hat{a}_j^{\perp} \\ \hat{b}_j \\ -\hat{b}_j^{\perp} \end{pmatrix} \begin{pmatrix} v - p_* \\ -(v - p_*)^{\perp} \end{pmatrix}^T \tag{4.16}$$

Figure 4.5 shows the tower deformed using this similarity-based method. In contrast to the affine method, the tower actually appears to be rotated, not sheared, to the left resulting in a more realistic deformation. Similarity transformations contain uniform scaling, which is apparent from the way in which the tower shrinks with the line segment. Rigid transformations remove this uniform scaling.

4.4.3 Rigid Lines

Using the solution from section 4.4.2 and Theorem 4.1, we immediately have a closed form solution for rigid-body transformations. The transformation matrix is, therefore, the same as equation 4.14 except we choose a different scaling constant μ_r so that $M^T M = I$.

$$\mu_r = \left| \sum_j \begin{pmatrix} \hat{a}_j^T & -\hat{a}_j^{\perp T} & \hat{b}_j^T & -\hat{b}_j^{\perp T} \end{pmatrix} W_j \begin{pmatrix} \hat{c}_j^T \\ \hat{d}_j^T \end{pmatrix} \right|$$

This deformation is non-linear, but we can compute it in a simple fashion using equation 4.8. This equation uses the rotated vector $\vec{f_r}(v)$, scales the vector so that its length is $|v - p_*|$ and translates by q_*. For this deformation using line segments, the rotated vector is given by

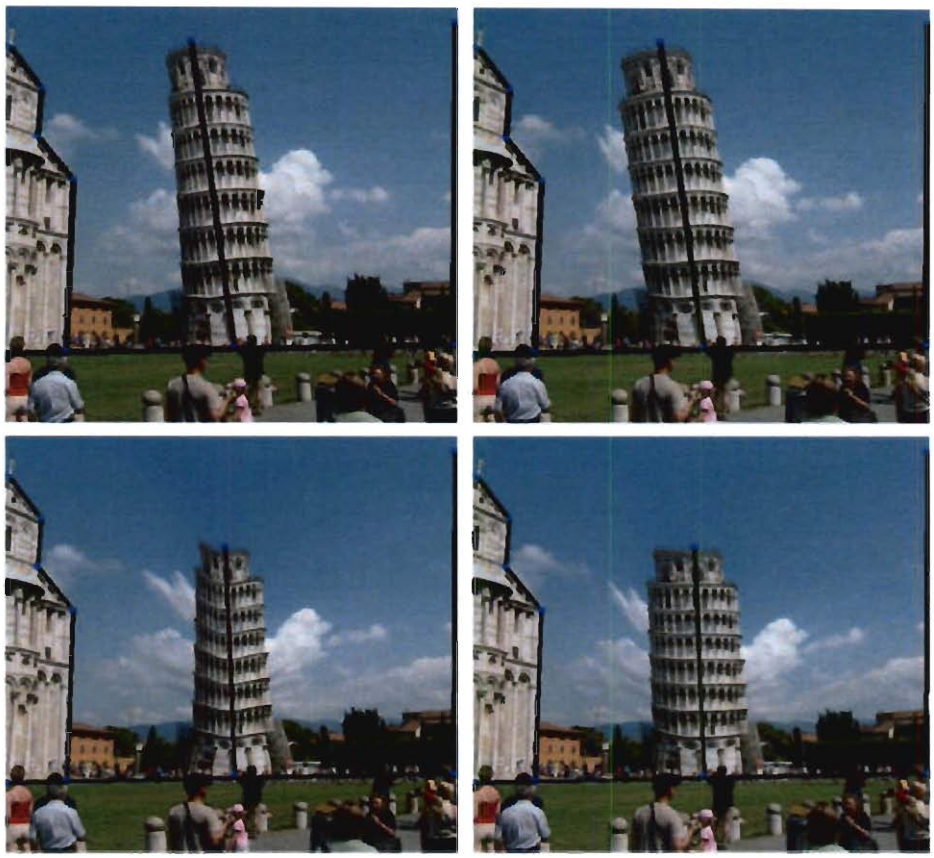


Figure 4.6 : Comparison of the line deformation method of Beier et al. (left) with the Rigid MLS deformation (right).

$$\vec{f}_r(v) = \sum_j \begin{pmatrix} \hat{c}_j & \hat{d}_j \end{pmatrix} A_j$$

where A_j is from equation 4.16.

Figure ?? (right) shows a deformation of the tower using this rigid method. In this deformation, the tower is rotated but does not shrink as the similarity deformation does. Instead the effect is almost the same as non-uniform scaling along the direction of the line segment.

Figure 4.6 also shows a comparison of the rigid deformation technique (right)

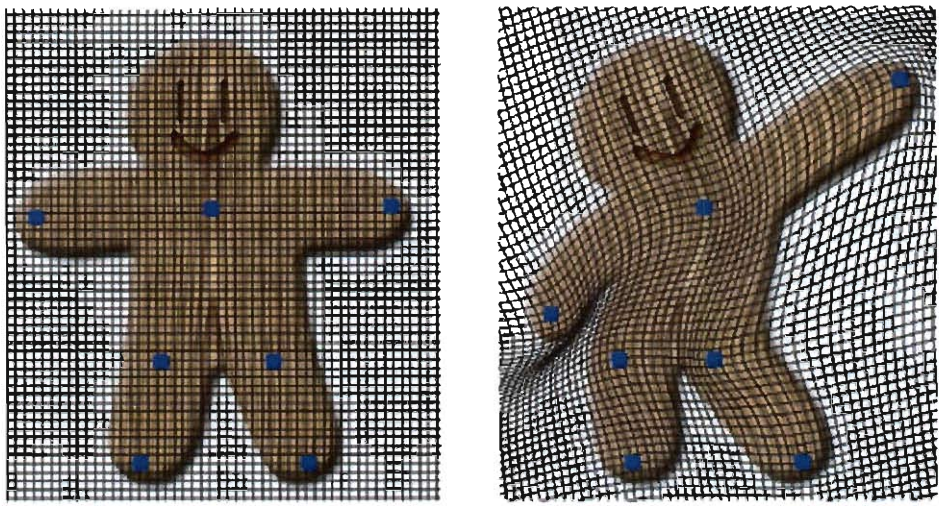


Figure 4.7 : Deforming an image with a uniform grid (50×50). Original image (left) and rigid MLS deformation (right) using bilinear interpolation in each quad.

with the line deformation method of Beier et al. [85](left). The warps created with Beier et al.'s method fold and pull in unrealistic ways whereas the rigid method does not suffer from these same defects.

4.5 Implementation

To implement these deformations, we precompute as much information as possible for the deformation functions $f(v)$. When we apply the deformation to an image, we typically do not apply $f(v)$ to every pixel in the image. Instead we approximate the image with a grid and apply the deformation function to each vertex in the grid. We then fill the resulting quads using bilinear interpolation (see figure 4.7).

In practice, this approximation technique produces deformations indistinguishable from the more expensive process of applying the deformation to every pixel in the image. For all of the examples in this paper, the images were approximately 500×500 pixels. To compute the deformations, we used grids on the order of 100×100 vertices.

Method	Figure 4.1 (7 points)	Figure 4.4 (11 points)	Figure 4.5 (7 lines)
Affine MLS	1.5ms	2.2ms	1.5ms
Similarity MLS	2.3ms	3.4ms	1.6ms
Rigid MLS	2.6ms	3.8ms	3.3ms
[84]	2.0ms	2.7ms	N/A
[85]	N/A	N/A	1.6ms

Table 4.1 : Table 4.1: Deformation times for the various methods.

If desired, more accurate deformations may be achieved with denser grids and the deformation time is linear in the number of vertices of these grids.

Table 4.1 shows the amount of time taken to deform each of the images using various methods on a 3 GHz Intel machine. Each deformation uses a grid of size 100×100. The rigid transformations take the longest due to the square root in the deformation function, but are still quite fast.

4.6 Conclusions and Future Work

We have provided a method for creating smooth deformations of images using either points or lines as handles to control the deformation. Using Moving Least Squares, we created deformations using affine, similarity and rigid transformations while providing closed-form expressions for each of these techniques. Though the least squares minimization with rigid transformations led to a non-linear minimization, we showed how these solutions could be computed directly from the closed-form deformation using similarity transformations thereby bypassing the non-linear minimization.

Figure 4.8 : Foldback caused during deformations.

In terms of limitations, our method may suffer from fold-backs like most other space warping approaches. These situations occur when the sign of the Jacobian of f changes. For many deformations, these fold backs may not be noticeable though extreme deformations will certainly cause such fold-backs to happen (see figure 4.8). For some deformations, fold-backs are acceptable, since these 2D images are meant to represent 3D objects. Igarashi et al. take advantage of the explicit topology of the image and provide a simple method for rendering these deformations. Our lack of topology makes this technique difficult though topological information may be added to our method.

In other applications, fold-backs are not desirable and must be eliminated. There is a generic approach available for fixing these fold-backs provided by Tiddeman et al. [96]. Given a warp, Tiddeman et al. create a subsequent warp such that the product of the two warps results in a non-negative Jacobian. Since we provide simple equations for our deformations, we intend to explore the possibility of constructing

closed-formed formulas for the Jacobian for use with Tiddeman et al.'s method.

Our warping technique also deforms the entire plane that the image lies in without regard to the topology of the shape in the image. This lack of topology is both a benefit and a limitation. One of the advantages of our approach is the lack of such topology, which creates a simple warping function. Other techniques such as Igarashi et al. [90] construct triangulations that outline the boundary of the shape and build deformations that depend on the specified topology. This topological information can create better deformations by separating parts of the images such as the legs of the horse in figure 4.4 that are geometrically close together. Notice that our method is general enough to accommodate different distance metrics dependent on the topology of the shape rather than the simple, Euclidean distance used as our weight factor. We intend to explore this issue in future work.

Finally, in the future we would like to explore generalizing these deformation methods to $3D$ to deform surfaces. Such a generalization has potential applications in motion capture where animation data can take the form of points in space for each frame of animation. However, the similarity transformation in section 4.3.2 no longer leads to a quadratic minimization, but an eigenvector problem and we are looking into methods to efficiently compute the solution to this minimization.

Chapter 5

Applying Volumetric Deformation to Lung Cancer Treatment Planning

In the previous chapter, we presented a technique for generating deformations from simple controls (points and lines.) In this chapter, we create a framework for deforming 3D volumetric data using these techniques. Furthermore, we apply this framework to lung cancer treatment planning. Within this application, we add custom features for lung segmentation of CT images as well as for establishing correlations between lung motion during deformation and lung ventilation, a measure used to quantity air circulation within a patient's lungs.

Deformation of volumetric data has many applications in medical imaging, image matching, fluid dynamics, and animation. Many standard approaches in these areas utilize automatic methods to generate deformations between sequences of images. However, automatically finding deformations for relatively noise-free images is an NP-complete problem [97, 98] because any algorithm can fall into local minima (see figure 5.1). The performance of such methods is worsened by the introduction of noisy images.

In radiation oncology, doctors determine the health of human lungs from analyzing single photon emission computed tomography (SPECT) ventilation images. Before the acquisition of a SPECT image, a patient drinks a liquid solution with radioactive tracer material that flushes through the patient's body. Afterwards, a SPECT scanner generates a 3D dataset where each voxel represents the time average concentration

of the radioactive tracer. SPECT images are coarse, forcing oncologists to make radiotherapy trajectories without fine metrics of the air flow within a patient.

This shortcoming has led to an area of research that infers lung ventilation from lung motion captured by image registration between pairs of 3D computed tomography (CT) images. Image registration is the process of deforming one image in a sequence of images to its successor in the sequence. This deformation captures the motion that the lungs undergo between the image pair. CT scanners can capture 3D datasets at much higher resolutions than that of SPECT scanners, yielding more accurate results. Higher image resolution allows oncologists to make more informed decisions on planning radiotherapy which can substantially lessen the amount of healthy tissue that is damaged by excessive levels of radiation. However, during the acquisition of CT images, substantial imaging artifacts can be generated due to CT gating error [99] (see figure 5.2). Gating noise introduces features in one image that are not present in the next image, which complicates generating accurate deformations between CT image pairs.

In fields such as oncology, extremely high-quality deformations are needed, and ultimately assessed by humans for quality assurance. For this reason, we have chosen to create semi-automatic methods and tools to aid humans in generating and validating deformations. Ideally, we want to create an environment that 1) deforms volumetric data at interactive rates and 2) gives real-time feedback for quality assurance.

5.1 Related Work

Substantial progress has been made in the area of volume visualization [100, 101, 102]. However, the problem of integrating volume manipulation and visualization needs more attention. Systems that alter and view volume data suffer from a combination

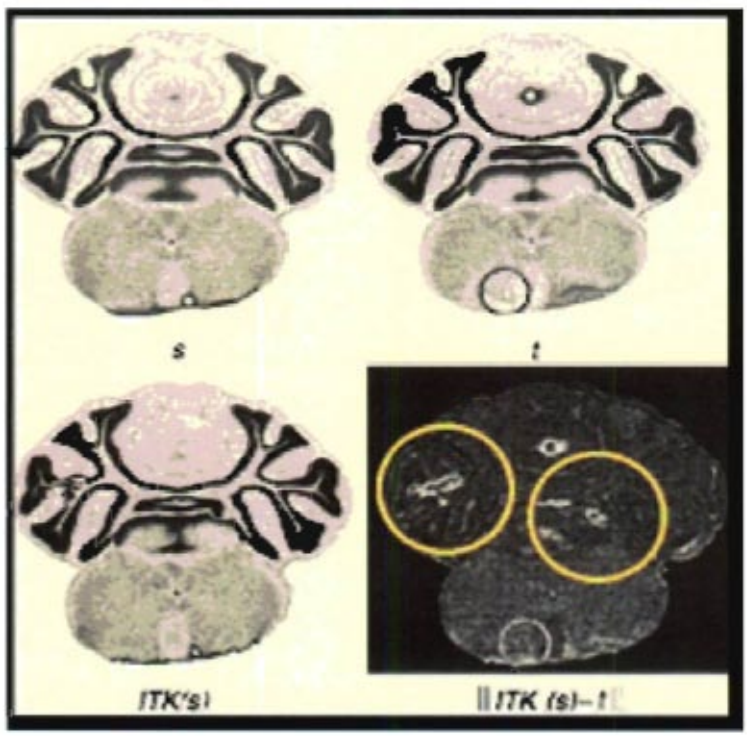


Figure 5.1 : Comparison of a source image, S, target image, T, the deformed image generated by ITK, and the difference between the deformed image and the target image.

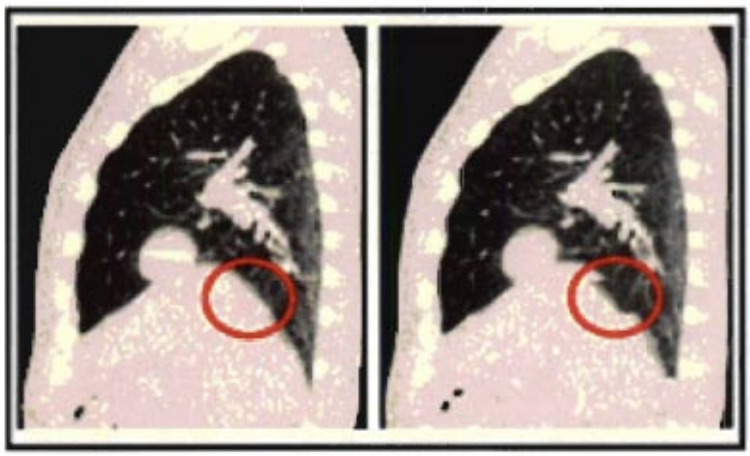


Figure 5.2 : Example of imaging artifacts due to gating error during the CT acquisition process.

of being either too cumbersome, non-intuitive for the user, or suffer from a lack of real-time interaction.

[103, 104]present techniques for generating volume deformations based on traditional skeletal animation. This work is natural for data which represents objects with rigid skeletal structure; however, limits its effectiveness on a wider class of images. Both works also do not facilitate real-time performance. [104] shows results for near interactive capabilities, however we desire a real-time solution.

[105, 106] have deformation frameworks that allow users to establish static correspondences between sets of landmarks for pairs of images, and induces deformations based on these correspondences. These environments have sub-real-time performance and use static landmark correspondences, not allowing users to manipulate these mappings interactively. [107] provides one of the first true interactive volume deformation environments. The controlling handles for this tool are curved wires which limit its effectiveness much like [104]. Additionally, this environment only deforms a source image and gives no metric to compare the result to a target image. It is also

not clear if this method can generate the high-quality deformations needed for our purposes while maintaining interactive speeds.

5.1.1 Volume Deformation

Substantial work has also been done in the realm of volume deformation. Automatic deformation methods based on conservation of energy equations such as Optical Flow [108] and modified versions such as Compressible Combined Local Global registration [109] are extensively used in the medical fields. These methods are computationally intensive, having run-times on the order of hours depending on the size of the deforming volume. Also, while these methods are theoretically sound, flow methods suffer from noise from real-world data which can yield unrealistic results (see figure 5.1).

Grid-based methods require generating a controlling grid which can be cumbersome to generate. For example, free-form deformations require an alignment of a lattice of control points with features in an image to generate C2 deformations using bivariate cubic splines. For 3D images, finite element methods require the generation of a tetrahedral mesh which can be time-intensive. In addition, this type of method is typically not well-suited for fast interaction which is needed for the purposes of our framework.

Due to their performance and intuitive manipulation, we use several techniques that can be organized in a manner similar to Shepard interpolants (i.e. $f(v) = \sum_i \alpha_i w_i q_i$). Due to the methods' simple structure, the CPU can process these deformations in real-time for data with a reasonable set of control handles, p_i(e.g. $200 < |p_i| < 700$).

5.2 Contributions

We provide an intuitive environment for generating and validating 3D deformations between pairs of volume data images. In order to deform volumes, we provide two existing point-based methods and create a new frame-based method for generating deformations between pairs of subsequent images, S and T. To aid in user control, our system automatically finds a set of landmarks in S and their corresponding displacements in T. Our main contribution is a rendering framework that facilitates interactive feature based deformations. At its core, we sample deformations on a uniform grid. This sampled deformation is stored as a RGB texture. We then use the synthesized texture to sample the respective volume data using conventional volume rendering techniques. This framework also provides real-time validation to measure the quality of the generated deformations.

We further customize our framework for manipulating breath-hold pairs lung CT images. We correlate the Jacobian of the generated 3D deformations to SPECT lung ventilation images in order to validate our results. Finally, we provide semi-automatic lobe segmentation for CT images for oncologist to perform lung computations on a lobe-basis.

5.3 Controlling Deformation

Our application generates deformations between pairs of input images. Before discussing the deformation techniques, we discuss automatically generating control handles in a source image and establishing respective displacements in a target image.

5.3.1 Handle Selection

Given a pair of 3D images, S and T, there are many approaches to producing deformations from S to T. A straightforward approach is to manually assign a displacement for each voxel in S. A displacement for a voxel v is a vector $d_v = f(v) - v$, where f is our deforming function). This approach is extremely accurate; however, manually producing displacements for each voxel in a 3D image is impractical for even moderate sized data. A more reasonable approach is to establish displacements for a subset of the voxels in S (300-500 voxels) and then create a mathematical function that blends these displacements to generate a deformation over all voxels in S. Even with this more tractable approach, the amount of work required can still be substantial. For this reason we have developed a computer-aided approach to select handles in a source image and to set the displacement of these handles in a target image.

We now consider the selection of handles that will control the deformation from a source image to a target image. We shall treat our source and target images as functions $S, T : N^3 \mapsto N$. Desirable control handles lie on features that are distinguishable in an image. These features lie on discontinuities, or edges of the images. To exploit this fact, we create $\tilde{S} = \Omega * S$ and $\tilde{T} = \Omega * T$ where $(*)$ is the convolution operator and $\Omega : N^3 \mapsto N$ is a Sobel operator. We then generate $\hat{S} = |S - \tilde{S}|$ and $\hat{T} = |T - \tilde{T}|$ to enhance the edges of S and T respectively. From this enhanced image, we generate a set of control handles by repeatedly finding the brightest voxel in the image and blackening a ball of user-specified radius around the selected voxel. The resulting set of handles corresponds to the strongest features in the image.

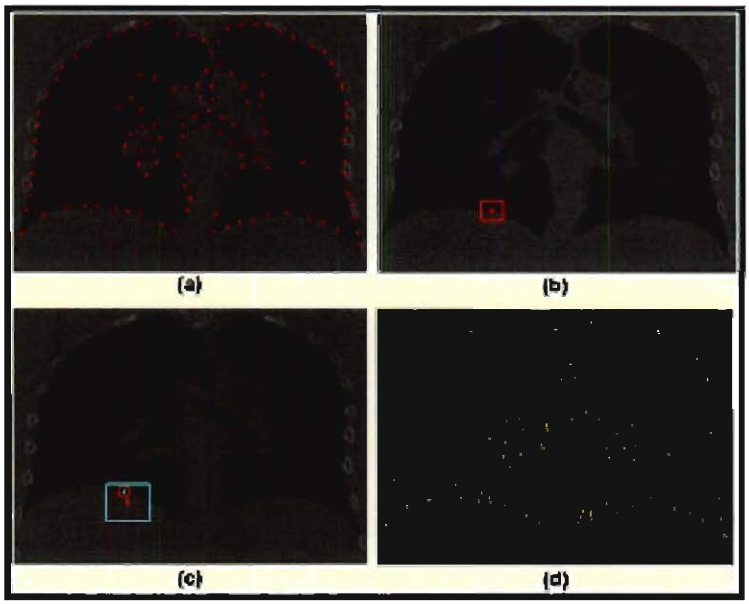


Figure 5.3 : Depiction of the automatic handle selection and correspondence process. (a) Source image with control handles determined by our selection algorithm. (b-c)Search for correspondence for one control handle, (d) The resulting correspondences for all of the handles.

5.3.2 Handle Correspondences

We move to establishing a mapping between the control handles p_i in S and their corresponding positions q_i in T. For each q_i, we perform the following:

- Initialize $q_i = p_i$ and generate 3D sub-images
- Center W_S and W_T about p_i and q_i
- Perform cross-correlation on W_S and W_T to determine the proper location of q_i

$$C(x, y, z) = \sum_i \sum_j \sum_k W_T[i, j, k] W_S[x + i, y + j, z + k]$$

In order to determine the proper mapping for handle q_i, we simply find the maximal value in C. This value corresponds to the best displacement that matches W_S to a window in T of the same size. However, there is no guarantee of a unique global maximum for C, so we create the function $\hat{C}$ that penalizes voxels that are far away from the control handle.

$$\hat{C}(x, y, z) = C(x, y, z) - \alpha|(x, y, z) - p_i|$$

Note that the size of W_T is substantially larger than that of W_S. This disparity enables our editor to compare the local neighborhood of a handle in a source image, and find the best match within the target image. For more accurate correspondences, we iterate this process several times. During every successive iteration, each new target window W_T is centered at the corresponding match, q_i, from the previous iteration. We also use the final correspondences to establish the best affine transformation centered at each control handle which is used by our frame-based deformation technique (described in the next section).

5.4 Deformation Techniques

We now discuss building deformations using a set of controlling handles, either in the form of points or coordinate frames. We treat a 3D deformation as a function that maps points in a source volume to points in a target volume. Imagine a set of control handles p_i associated with S that are moved to q_i associated with T. Now in order for a deformation function to provide meaningful results, there are several properties that the deformation must satisfy:

- Smoothness: f should smoothly deform the volume

- Identity: If $\forall i$, $p_i = q_i$ then $\forall v$, $v = f(v)$
- Interpolation or Approximation: Either $f(p_i) = q_i$ or $|f(p_i) - q_i| < \epsilon$

In order to generate real-time deformations, our framework assumes that the handles for the source image are fixed at the time the deformations are generated. We exploit this property and precompute information to facilitate realtime manipulation. If a user alters the source handles, the realtime deformation computation is halted due to the necessary precomputations (details on the performance delay is in section 5.5.2). One of the intended uses of our editor is to manipulate q_i in order to generate interactive deformations facilitated by these precomputations.

All of our techniques reduce the deformation of a voxel to the form: $f(v) = \sum_i \alpha_i q_i$ or $f(v) = \sum_i \alpha_i q_i + \alpha_{ix} q_{ix} + \alpha_{iy} q_{iy} + \alpha_{iz} q_{iz}$. Here f is the deformation function, α_i are scalar coefficients, and q_i are the displaced handles in the target image. Expressing deformations as a weighted combination of the deformed control handles is the key feature that allows our application to generate deformations at interactive speeds.

5.4.1 Thin Plate Splines

The first technique we describe to generate deformations is thin-plate splines. We will proceed with a brief description of the theory behind thin-plate splines [84]. For a more in depth look into thin-plate splines, refer to [84]. A d-dimensional thin-plate spline is the fundamental solution to the biharmonic equation with the form

$$\hat{f}(v) = U(r) = r^2 lnr$$

where $v \in \mathbb{R}^d$ and $r = |v|$. At the root of thin-plate spline theory is the physical notion of bending a thin sheet of metal represented by lofting $U(x, y)$ over the xy-

plane. Here the physical lifting is orthogonal to the plane. To apply thin plate splines to the problem of interpolating scattered control points, one treats the lifting of the plate as the actual displacement of the individual components of the control points. We now wish to generate a function $f(x, y, z)$, comprised of scaled translates of our thin-plane spline basis function $\hat{f}$, such that $f(x, y, z)$ minimizes the bending energy given by

$$\iiint_{\mathbb{R}^3} f_x x^2 + f_y y^2 + f_z z^2 + 2(f_x y^2 + f_x z^2 + f_y z^2) dx dy$$

The thin-plate algorithm reduces to a system of linear equations that are directly solved in [84]. We slightly reorganize the approach in [110] to generate a deformation of the form $f(V) = \sum_i \alpha_i q_i$.

5.4.2 Moving Least Squares Deformation

Next, we discuss generating a deformation from S to T using a slightly modified affine moving least squares approach from the previous chapter. To recap, given a voxel v in S, we want to find the best affine transformation, A_v, which minimizes

$$\sum_i w_i |A_v(p_i) - q_i|^2$$

Here, p_i and q_i are the control points in $\mathbb{R}^3$ and w_i is either $w_{i_{interp}} = \frac{1}{|p_i - v|^{2\alpha}}$ or $w_{i_{approx}} = \frac{1}{|p_i - v|^{2\alpha} + \epsilon}$ where $\epsilon > 0$.

With this approach, our deformation function becomes $f(x, y, z) = A_v(x, y, z)$. For brevity, we proceed to a closed form solution for this minimization problem. For a complete derivation of this method, refer to the previous chapter. After manipulation, we can rewrite the minimization problem as

$$\sum_i w_i |\hat{p}_i M - \hat{q}_i|^2$$

Here $\hat{p}_i = p_i - \frac{\sum_i w_i p_i}{\sum_i w_i}$ and $\hat{q}_i = q_i - \frac{\sum_i w_i q_i}{\sum_i w_i}$ are weighted centroids.

After we use the solution to the classical normal equation to determine the minimizing M, we can rewrite our deformation function as $f_v(x, y, z) = \sum_j \alpha_j q_j$. Here α_j is determined by

$$\alpha_j = \Lambda_j + \frac{w_j(1 - \Lambda_j)}{\sum w_k k}$$

and Λ_j is a scalar that can be determined by

$$\Lambda_j = (x - p_*)(\sum_i \hat{p}_i^T w_i \hat{p}_i)^{-1} w_j \hat{p}_j^T$$

where $p_* = \frac{\sum_i w_i p_i}{\sum_i w_i}$.

Observe that although the points q_i can change, the points p_i are fixed and allow Λ_j to be precomputed. Thus, we can predetermine α_j reducing our deformation function f to a summation of scalar multiples of the points, q_i.

5.4.3 Frame-based Moving Least Squares Deformation

We now extend the moving least squares framework to generate deformations by manipulating coordinate frames as opposed to control points. Let the set of coordinate frames in our source image be $[p_i, p_{i_x}, p_{i_y}, p_{i_z}]$ and their corresponding matches in our target image be $[q_i, q_{i_x}, q_{i_y}, q_{i_z}]$. Here, $[p_i, p_{i_x}, p_{i_y}, p_{i_z}]$ and $[q_i, q_{i_x}, q_{i_y}, q_{i_z}]$ are the local origins, x-vectors, y-vectors, and z-vectors for the respective coordinate frames. Now given a voxel v in our source image, we want to find the best affine transformation, A_v, which minimizes a modified version of equation 5.4.2.

$$\sum_i w_i |A_v(p_i) - q_i|^2 + \beta(|p_{ix}M - q_{ix}|^2 + |p_{iy}M - q_{iy}|^2 + |p_{iz}M - q_{iz}|^2)$$

were β is a scalar which regulates the influence of the vector constraints in our new minimization problem. Note that $\beta = 0$ reduces our expression to equation 5.4.2.

As before, we define our deformation function $f_v(x, y, z) = \sum_j A_v(x, y, z)$. Observe that after expanding the quadratic represented by equation 5.4.3, we have an expression that is still quadratic in T and has no new terms involving T. This enables us to use the same solution for T in terms of M described in equation 5.4.2. Substituting T in equation 5.4.3 allows the minimization problem to be expressed as

$$\sum_i w_i |\hat{p}_i M - \hat{q}_i|^2 + \beta(|p_{ix}M - q_{ix}|^2 + |p_{iy}M - q_{iy}|^2 + |p_{iz}M - q_{iz}|^2)$$

As in the original moving least squares method, we use the solution to the normal equation to find M that minimizes equation 5.4.3

$$M_* = \sum_i \hat{p}_i^T w_i \hat{p}_i + \beta(\hat{p}_{ix}^T w_i \hat{p}_{ix}^T + \hat{p}_{iy}^T w_i \hat{p}_{iy}^T + \hat{p}_{iz}^T w_i \hat{p}_{iz}^T)^{-1} \tag{5.1}$$

$$M = M_* \sum_j w_j (\hat{p}_j^T \hat{q}_j + \beta(\hat{p}_{jx}^T \hat{q}_{jx}^T + \hat{p}_{jy}^T \hat{q}_{jy}^T + \hat{p}_{jz}^T \hat{q}_{jz}^T)) \tag{5.2}$$

With our closed-form solution for M, we can now write a simple expression for our deformation function $f_v(x, y, z)$.

$$f_v(x, y, z) = \sum_j \alpha_j q_j + \Lambda_{jx} q_{jx} + \Lambda_{jy} q_{jy} + \Lambda_{jz} q_{jz}$$

where Λ_{jx}, Λ_{jy}, Λ_{jz}, and α_j are precomputed scalars determined by

$$\Lambda_j = (x - p_*) M_* \hat{p}_j^T \tag{5.3}$$

$$\Lambda_{jx} = (1, 0, 0) M_* \hat{p}_{jx}^T \tag{5.4}$$

$$\Lambda_{jx} = (0, 1, 0) M_* \hat{p}_{jy}^T \tag{5.5}$$

$$\Lambda_{jx} = (0, 0, 1) M_* \hat{p}_{jz}^T \tag{5.6}$$

$$\alpha_j = \Lambda_j + \frac{w_j(1 - \Lambda_j)}{\sum_k w_k} \tag{5.7}$$

Note the constant vectors $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ constrain our source handles to be axis aligned.

5.5 Visualization and Quality Validation

5.5.1 Volume Rendering

Volumes are typically viewed either by extracting isosurfaces in the form of surface meshes or by directly rendering the volume. Contouring algorithms such as marching cubes [16] or dual contouring [14] are used to create surfaces from volume data; however this approach is not well suited for viewing interior information. One of the more widespread direct volume rendering methods is texture-based visualization [100, 101, 102].

In texture-based methods, volume data is commonly represented as a regular grid of values that sample a function (e.g. a scalar field $\tilde{V} : \mathbb{R}^3 \mapsto \mathbb{R}$). This data is treated as a texture $\tilde{V} : [0, 1]^3 \mapsto [0, 1]$ and stored on a graphics card. With this texture information, the standard approach to visualizing the volume requires generating a dense sequence of planes parallel to the image plane and intersecting them with the unit cube $[0, 1]^3$. The intersections are treated as polygons with the texture

coordinates of each vertex set equal to its position (i.e. $tex(x, y, z) = \{x, y, z\}$). These polygons are processed by a vertex shader and the texels of these polygons are processed by a transfer function implemented on a pixel shader.

Viewing deformed volumes imposes an additional challenge. With present hardware capabilities, realtime calculation of deformed volume sets is not feasible. However, using our feature-based techniques, interactively sampling the deformation on coarser grids is achievable.

5.5.2 Rendering Framework

In order to have an effective tool for validating and generating deformed images, a user should be able to:

- View subvolumes around selected pairs of control handles in order to assess the quality of the displacements (used for generating deformations).
- Visually determine the global quality of the deformed target volume (generated by our framework) with respect to the undeformed source volume.

We accomplish this by uniformly sampling the generated deformations as a RGB texture which is passed to the GPU. Using a sweeping algorithm (see [111]), we create a set of planes that intersect the cube from front to back. As the graphics card processes a polygon, a pixel shader uses trilinear blending to approximate the deformation function using the previously mentioned RGB texture for each processed fragment. Using this synthesized texture coordinate, the pixel shader then samples the data being visualized.

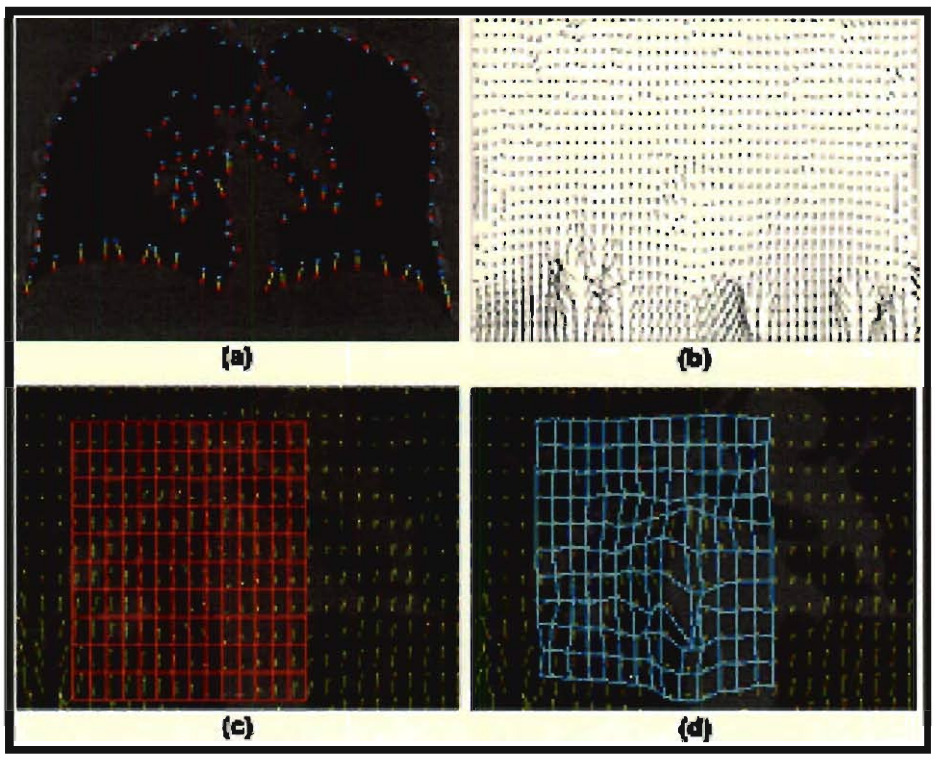


Figure 5.4 : 2D depiction of viewing deformations using our framework. (a) Source image with control handles and displacements. (b) Deformation field from S to T. (c-d) Regular and deformed grid used to sample S and T.

5.5.3 Local Comparison of Correspondences

Now consider locally comparing the handle correspondences used in our deformation methods. Our goal is to overlay a sub-cube, $cube_{source}$, of the source image centered at the source handle and a sub-cube, $cube_{target}$, of the target image centered at the target handle. Assume the dimensions of $cube_{source}$ and $cube_{target}$ are identical. For each node in our uniform grid, we store a texture coordinate in our synthesized target image for $cube_{target}$ (i.e. $tex(x, y, z) = A_v(x, y, z)$). Here $A_v(x, y, z)$ is a thin plate spline or best affine for q_i as described in section 5.4. Afterwards, we perform a normal volume rendering pass for $cube_{source}$. Then using our generated target texture coordinate space, we visualize $cube_{target}$ as described in the previous paragraph. This overlays the two sub-cubes around the handles, p_i and q_i in the same cube.

5.5.4 Viewing Deformations

Our last goal is to globally determine the quality of the deformed target image in real-time. In order to exactly view the entire deformed volume, our application would need to store and compute the sum of the scaled control handles for each voxel in the deformed image. Realtime performance for this approach is impractical, even for modestly sized data sets. To compensate, we generate an approximation of the deformed image using the rendering framework mentioned earlier.

When comparing the quality of our deformation, note that the image $f(S)$ is not a surjective mapping from S to T. In light of this, we must compare our source image, S, with our deformed image $f^{-1}(T)$. We begin by altering the content of the previously mentioned synthesized target image. At each node in our uniform grid, we now store weights corresponding to each control handle. Now whenever a user moves a handle in the target image, we recompute a deformed texture coordinate for each

node in our uniform grid. We then update the synthesized target image with these deformed coordinates and render both volumes as mentioned in the local comparison section. This results in a trilinear approximation to $f^{-1}(T)$ (see figure 5.4).

Due to the nature of the deformations, trilinear blending yields high-quality approximations when the grid is sampled to a reasonable resolution. In order to maintain realtime calculations with a dense set of sample points, we exploit the fact that $f(v) = \sum_i \alpha_i q_i$ to speed up the deformation calculation. Whenever a user selects a pair of handles (p_i, q_i) to alter, we store $\hat{f}(v) = \sum_{j \neq i} \alpha_j q_j$ for each sampled vertex. Storing $\hat{f}(v)$ enables our application to modify the synthesized target coordinate space with a simple multiplication and addition for each node in the grid. To further aid in the time costs of our method, we only recompute the weights in the grid when the source handles are generated. The necessary precomputations of the weights are on the order of seconds for our provided methods (Results are given in section 5.8).

5.5.5 Quality Assessment

Manually comparing two overlaid volumes can be tedious. In light of this, our framework gives aid in quality validation via a transfer function provided by one of the provided pixel shaders. Transfer functions are ran on fragments from the graphics card and return an rgba color tuple to be output for viewing. When assessing the quality of matching volumes, a transfer function should be:

- Symmetric (i.e. $h_{TtoS}(x, y, z) = h_{StoT}(x, y, z)$
- Penalize for divergent values

We provide pseudocode for the transfer function used by our framework (see algorithm 5.1). As can be seen in the pseudocode, the color returned by this function is

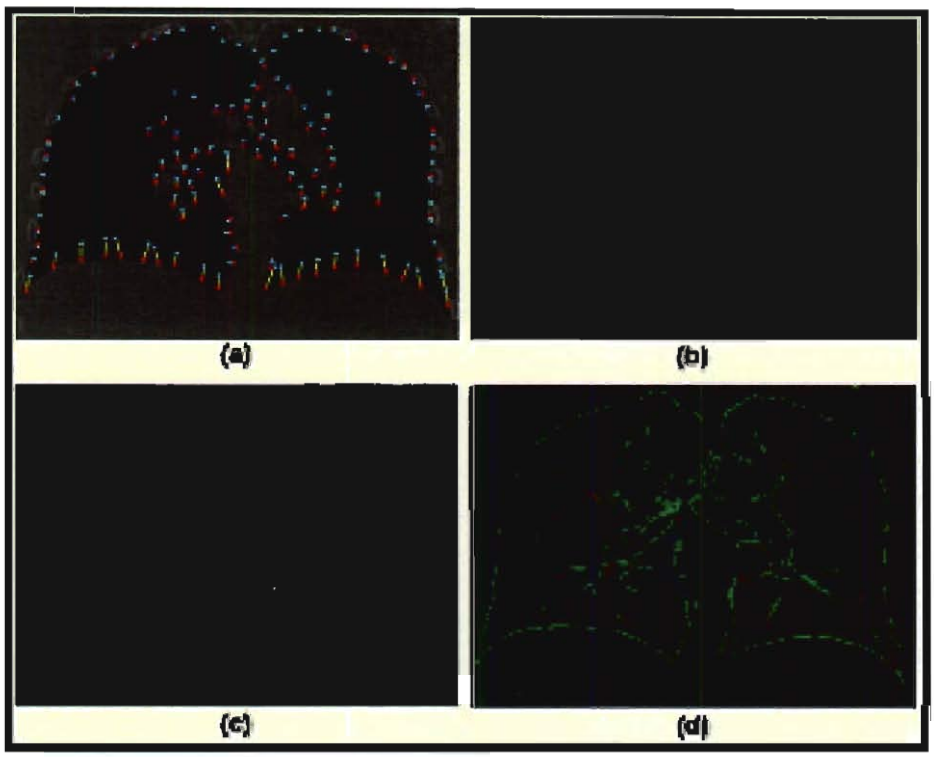


Figure 5.5 : 2D depiction of the quality assessment feature of our application. (a)Source image with control handles and displacements. (b)Target image. (c)Comparison of S and T. (d) Comparison of S and $f^{-1}(T)$ (Note the number of blue or red areas is lessened and major features are better matched than in (c).

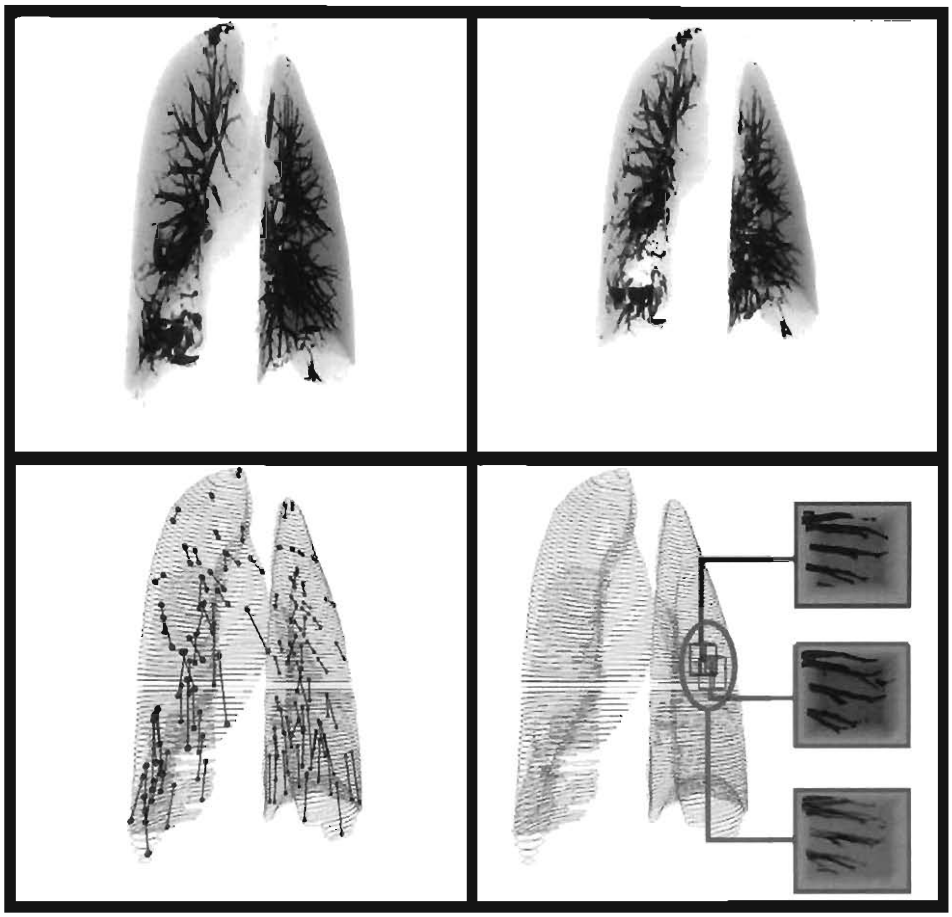


Figure 5.6 : Series of images from our application (left to right): Source Volume, Target Volume, Correspondences, and Comparisons of local volumes

green if there is a match and either red or blue if either the source or target image has greater intensity at the input location. This function also changes the opacity linearly in regards to the magnitude of the intensity difference between the two volumes at a given texel. The rightmost image in figure 5.6 gives a 3D example of this quality assessment.

Algorithm 5.1 Assessing Quality of $f^{-1}(T)$ versus S

1: $(u_S, v_S, w_S) = tex(\Omega_S, (x, y, z))$
2: $(u_T, v_T, w_T) = tex(\Omega_T, (x, y, z))$
3: $i_S = tex(S, (u_S, v_S, w_S))$
4: $i_T = tex(T, (u_T, v_T, w_T))$
5: **if** $i_S \neq 0 \wedge i_T \neq 0$ **then**
6: $\quad \gamma = min(i_S, i_T)$
7: $\quad col.rgb = (i_S - \gamma, 2\gamma, i_T - \gamma)$
8: $\quad$ **return** $col\dot{\beta}$
9: **end if**
10: **return** $(0, 0, 0, 0)$

5.6 Construction of Lobe-based Lung Model

We next consider constructing a model of the lobe boundaries from a single medium-resolution (approximately 1 mm^3) 3D CT of the lungs. Our approach consists of four simpler steps

- Extract the outer boundaries of the lungs using a simple combination of thresholding and majority filtering of 3D images
- Process 3D lung images to enhance the interior lobe boundaries
- Extract the 2D cross-sections of the lobe boundaries from this enhanced image via a semi-automatic method based on dynamic programming

To illustrate these steps, figure 5.7 shows a typical horizontal 2D cross-section of a 3D CT image of a patients lungs being processed.

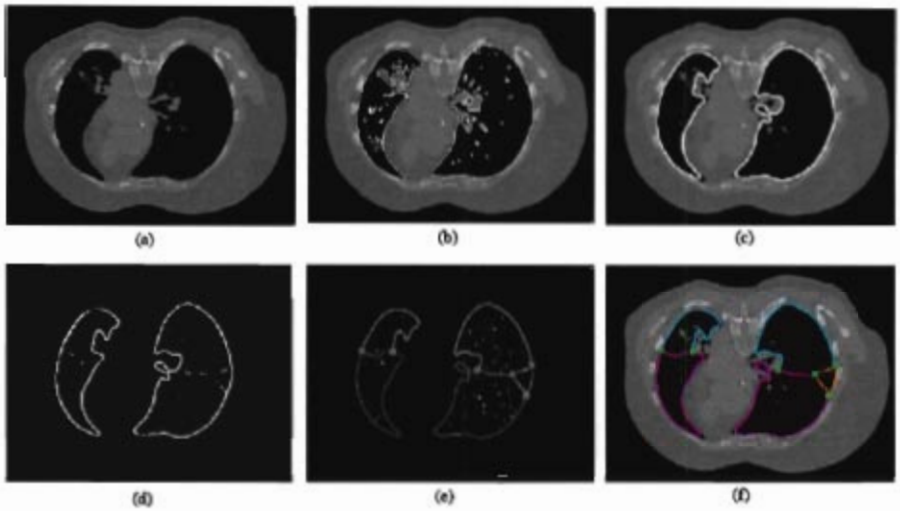


Figure 5.7 : Original 2D cross section (a). Contour of low and high density voxels (b). Outer boundary after max filter pass (c). Lungs with enhanced lobes (d). Traced paths between user points (e). Lobes overlaid on cross section (f).

5.6.1 Outer lung boundary construction

We begin by computing a rough approximation to the outer boundary of the lungs by thresholding the raw CT data into a binary image consisting of low-density voxels (black) and high-density voxels (white). Figure 5.7b shows the 2D contour that separates these two sets of voxels on our typical cross-section.

While this contour does include two large connected volumes of low-density voxels corresponding to the lungs, the contour includes numerous extra components due to the presence of high-density voxels that model pulmonary arteries and other higher-density anatomical structures interior to the lungs.

To smooth these two volumes and eliminate interior white voxels, we apply several rounds of *majority filtering*. Each round of majority filtering recolors a voxel based on the majority color of the $3 \times 3 \times 3$ block of surrounding voxels. This filtering erodes noisy features on the outer boundaries of the lungs and eliminates small features from

the interior of the lung volumes. The curve in figure 5.7c is a 2D cross-section of this smoothed volume.

5.6.2 Interior lobe boundary enhancement

As is apparent in figure 5.7a, the interior boundaries of the lobes are faint edges that are barely visible, even to the expert eye, in 2D CT cross-section. To aid in the accurate reconstruction of these lobe boundaries, we first enhance these images to emphasize the interior lobe boundaries.

To this end, we filter the 3D image based on locally fitting a quadratic function to the image. At a particular voxel, we fit (in a least squares sense) a quadratic function of the form

$$q(x, y, z) = a + bx^2 + 2cxy + 2dxz + ey^2 + 2fyz + gz^2$$

to the $5 \times 5 \times 5$ neighborhood of intensity values centered at this voxel. (For the sake of simplicity, we assume that the voxel in question is at the origin.) Next, we construct the 3×3 matrix corresponding to the pure quadratic portion of $q(x, y, z)$,

$$\begin{pmatrix} b & c & d \\ c & e & f \\ d & f & g \end{pmatrix}$$

.

Finally, we compute the dominant eigenvalue of this matrix. If the dominant eigenvalue is negative, the fitting quadratic has a primarily decreasing parabolic shape. Near a lobe boundary, the peak of this parabola aligns with the lobe boundary with parabola decreasing towards the low-intensity regions adjacent to the lobe boundary.

Our classification method consists of determining if the dominant eigenvalue is negative and testing whether the voxel intensity lies in the range $(0.05, 0.08)$ typical of voxels on the lobe boundary. (Here, the voxel image intensities are normalized to be in the range from 0 to 1.) Figure 5.7d shows the effect of applying our filter to a typical cross-section of a 3D CT image.

5.6.3 Interior lobe boundary construction

The final task for this step is to construct the actual lobe boundaries from the enhanced images. To this end, we propose a semi-automatic method for constructing the interior boundaries on a cross-section by cross-section basis. Given the outer lobe boundaries for a horizontal cross-section, the user places two points on the outer boundary of the lungs and the system automatically traces a path between these two points by following potential lobe pixels in the enhanced image. This path following process can be easily automated by using a standard shortest path algorithm such as Dijkstra's algorithm [112]. In the case where three lobes meet at a point (such as in figure 5.7e), the user may create each of the three interior boundary curves meeting at this point separately.

Given these interior boundary curves, we then combine the interior and exterior boundary curves to form closed curves bounding each separate lobe in the cross-section. These regions are then automatically labeled as corresponding to a particular lobe based their rough anatomical position.

To compute a complete lobe-based model of the lungs, the user cycles through the various horizontal cross-sections comprising the lungs. Since there are hundreds of such cross-sections, we allow the user to inherit the interior lobe geometry (and topology) from one processed section onto an adjacent cross-section. Typically, the

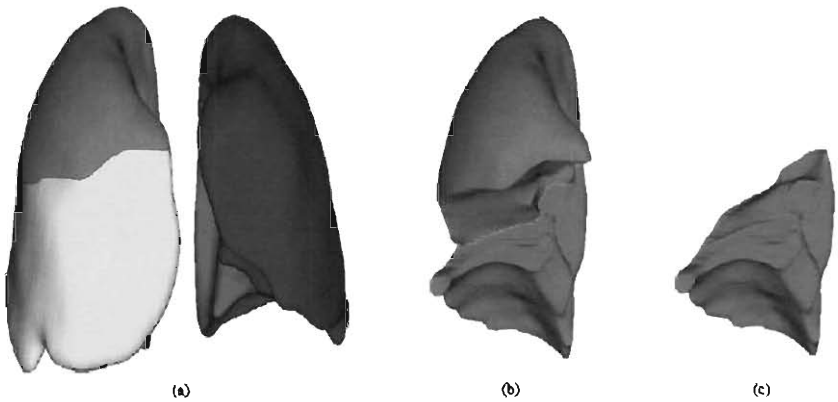


Figure 5.8 : Lobe-based surface model(a). Subset of right lobes (b) and (c).

user makes some minimal adjustment of the endpoints of the interior boundary curves that requires on the order of 5-10 seconds of attention. The result of this process is that an experienced user can assemble an accurate lobe-based model of the lungs in a timeframe on the order of 15-20 minutes. Given this sequence of curve networks for horizontal cross-sections, we may optionally assemble this sequence of curve networks into a surface network that partitions the lungs into lobes [113]. Figure 5.10 shows such a lobe-based surface model. Due to our need to visualize the interior of the lungs later, we typically render the outer lung shape using only the curve model.

One might wonder whether using more sophisticated 3D deformable model technology would have allowed us to avoid the need to manually construct the interior lobe boundaries. In our experience, the anatomical variation in lobe geometry between various patients lungs (especially diseased lungs) makes constructing an accurate 3D atlas suitable for applying traditional 3D deformable modeling techniques to this problem difficult

5.7 Estimation of ventilation and perfusion from 3D deformations

During breathing, the lung tissue deforms due to the movement of both air and blood in and out of the lung tissue. Given the deformation $f(x)$ that warps an inhale CT image I onto an exhale CT image E, our main goal is to estimate the local airflow and blood flow in the lungs from the two images and the deformation.

The key to this estimation is to compute the change in volume induced by the deformation $f(x)$. Given point x, let B_x be a small box around x in the inhale image. Let $f(B_x)$ be the corresponding deformed box. Now, we defined an associated function $jac(x)$ of the form

$$jac(x) = \frac{vol(f(B_x))}{vol(B_x)}$$

.

Mathematically, the function $jac(x)$ is known as the *Jacobian* of the deformation $f(x)$ [114]. Figure 5.9b shows a plot of $jac(x)$ for our typical cross-section.

5.8 Results

We performed a series of tests to assess the performance of different components of our application. These results were taken on a PC with an Intel(R) Xeon 3.2GHz processor with 4GB RAM and a GeForce 8800 GTX graphics card. We used three test images: CT5 (a CT image of a human lung), bunny (a CT image of the Stanford bunny), and brain (an MRI image of a human brain). In all cases, the grid used to synthesize our textures had dimensions $32 \times 32 \times 32$.

All timing results are measured in seconds except for the frame rate which is

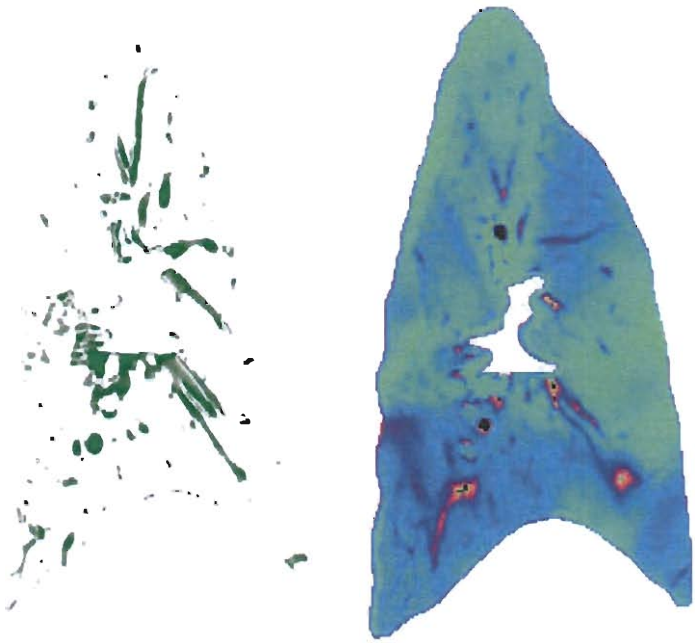


Figure 5.9 : Perfusion image to the left (a) and ventilation image to the right (b).

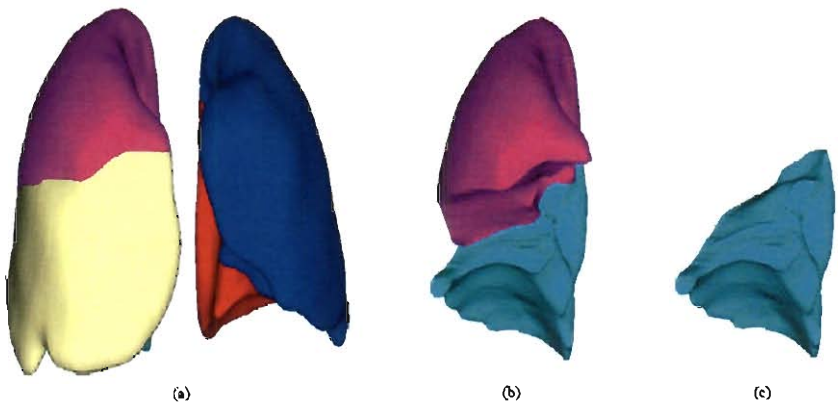


Figure 5.10 : Lobe-based surface model(a). Subset of right lobes (b) and (c).

			Setup Times		
Dataset	Size	No. of handles	MLS	MLS Frame	TP
CT5	256x256x112	200	1.501	1.703	13.22
Bunny	512x512x360	200	0.781	0.907	5.13
Brain	256x256x109	216	1.36	1.563	15.50
CT5	256x256x112	312	1.969	2.297	35.33
Bunny	512x512x360	343	2.172	2.516	44.11

Table 5.1 : Setup Times for Generating Deformations

measured in frames/sec. From our results, the bottleneck of our software application is processing the weights for our control handles for our different deformation methods. Of the three methods, our thin-plate spline implementation suffers from large setup times as the number of control handles increase. The other two deforming algorithms based on a moving least squares approach have reasonable setup costs. Note that once the weight computation is complete, the time required to alter the deformations is negligible in comparison to the time needed to render a single volume image. For all of the frame rate measurements, the volume encompassed a 900×900 $pixel^2$ screen area.

5.9 Conclusion and Future Results

We have provided an editor for deforming pairs of 3D images. Our editor automatically generates a set of controlling handles on features within a 3D source image. It determines correspondences for these handles in a subsequent target image. Our application then uses these correspondences to generate deformations via several

			Setup Times			
Dataset	Size	No. of handles	MLS	MLS Frame	TP	FPS
CT5	256x256x112	200	0.62ms	0.31ms	0.31ms	69.1
Bunny	512x512x360	200	0.78ms	0.32ms	0.32ms	68.5
Brain	256x256x109	216	0.63ms	0.47ms	0.47ms	69.4
CT5	256x256x112	312	0.63ms	0.47ms	0.47ms	68.9
Bunny	512x512x360	343	0.78ms	0.47ms	0.47ms	68.3

Table 5.2 : Interaction Speed of Generated Deformations

handle-based methods. Further manipulation of these handles induces changes in the generated deformation which is reflected in realtime.

The framework also provides visualization methods for realtime fidelity assessment of local handle correspondences and global deformation matching.

As far as limitations, our tool currently does not model non-smooth motion. For example, our editor could not create a deformation that captured the motion of a brick sliding across another brick. This limitation is due to the smoothness property of the deformations generated by this framework. One way to approach this problem in the future is to give the user the ability to paint, or segment, the various portions of the volumes and tag handles to affect only voxels of particular colors. This painting feature would allow for discontinuities to exist on the boundaries of different segmented regions. We would also like to address fill-rate limitations of our rendering framework with new methodologies for empty-space culling. There exists methods that address this problem [115]; however these methods do not facilitate our visualization framework. We also want to synthesize finer texture coordinate images (i.e. 64^3 and 128^3 grids); however for moderate data, we would need to improve the setup

times for our deformation methods with parallel computing.

Chapter 6

Conclusion

Throughout the last four chapters, we have developed ways to efficiently view volumetric data, edit 3D surfaces with an hybrid implicit approach, deform 2D/3D images with volumetric functions; and apply 3D volumetric deformation techniques, dynamic programming on volumetric grids, and contouring techniques to lung cancer treatment planning. These tools all have advantages and disadvantages. For instance, the cube cutting method is used only in texture-based volume rendering. While the class of rendering techniques is limited for Cube Cutting, there are substantial fields that primarily view grids of scalars without the need of realistic lighting provided by techniques such as raytracing and raycasting. Furthermore, the Cube Cutting method renders polygons for texturing faster than any method that has been published thus far.

The Moving Least Squares framework creates deformations without the need for topology information present in previous works. However, controlling fine details with point deformations can require many points due to the lack of topological information. All the same, these point deformation techniques create a framework that can easily change the class of deformations used from rigid to similarity to affine to thin-plate deformations.

Applying the Moving Least Squares framework to lung cancer treatment planning generates 3D functions that faithfully approximate the motion of the lungs during patient breath cycle. The results from this work have also been used as

the gold standard for comparing the results from automated deformation techniques (see http://www.dir-lab.com). There are also preliminary results validating the correctness of these deformations from comparisons between the Jacobian and SPECT ventilation images.

Finally, Depth Peel Editing offers an implicit representation of surfaces that can be used to perform boolean and smoothing operations. Furthermore, this representation can be used to contour surface meshes with greatly improved runtimes. However, a major bottleneck of this approach is the time used to pass information from the GPU to the CPU.

Depth Peel Editing actually hints at the future areas of research. Moving all Depth Peel Editing computations to the GPU relaxes many of the current grid size limitations of CSG, smoothing, and contouring techniques. However, there are several obstacles to consider. We would need a surface representation that can efficiently facilitate surface changes from the GPU. We also would want to generate depth peel images in one direction in one rendering pass, regardless of the depth complexity of the shape. This either requires nVidia to allow output into 3D textures or for a CUDA implementation of portions of the rasterization features offered with the normal OpenGL rendering pipeline. Furthermore, allowing the user to edit the mesh while changing the view of the scene requires viewing our implicit representation from oblique directions. There may be techniques that can be adapted from light field rendering.

Exploiting the power of GPU/CPU hybrid systems is the clear direction for future endeavors. Shifting computation for Cube Cutting, deformation calculations from Moving Least Squares, and dynamic programming algorithms to the GPU will substantially increase productivity in many fields. Chapter 5 provides a great example

of the possibility of applications in non-graphics fields.

Bibliography

[1] C. Rezk-Salama and A. Kolb, "A Vertex Program for Efficient Box-Plane Intersection," in *Proc. Vision, Modeling and Visualization (VMV)*, pp. 115 122, 2005.

[2] G. D. Rubin, C. F. Beaulieu, V. Argiro, H. Ringl, A. M. Norbash, J. F. Feller, M. D. Dake, R. B. Jeffrey, and S. Napel, "Perspective volume rendering of CT and MR images: applications for endoscopic imaging.," *Radiology*, vol. 199, no. 2, pp. 321 330, 1996.

[3] P. S. Calhoun, B. S. Kuszyk, D. G. Heath, J. C. Carley, and E. K. Fishman, "Three-dimensional Volume Rendering of Spiral CT Data: Theory and Method1," *Radiographics*, vol. 19, no. 3, pp. 745 764, 1999.

[4] M. Matsumoto, N. Kodama, Y. Endo, J. Sakuma, K. Suzuki, T. Sasaki, K. Murakami, K. Suzuki, T. Katakura, and F. Shishido, "Dynamic 3D-CT Angiography," *AJNR Am J Neuroradiol*, vol. 28, no. 2, pp. 299 304, 2007.

[5] K. Takizawa, C. Moorman, S. Wright, J. Purdue, T. McPhail, P. Chen, J. Warren, and T. Tezduyar, "Patient-specific arterial fluid-structure interaction modeling of cerebral aneurysms," *International Journal for Numerical Methods in Fluids*.

[6] D. M. Chen and C. Musat, "Fast Seismic Volume Rendering in a Multiresolution Framework," *AGU Fall Meeting Abstracts*, pp. B403+, Dec. 2007.

[7] T. U. C. C.-k. D. S. L. T. U. Y. J. M. H. T. U. Lin, Jim Ching-rong (Sugar Land, "Systems and methods for imaging a three-dimensional volume of geometrically irregular grid data representing a grid volume," December 2009.

[8] T. Tu, H. Yu, L. Ramirez-guzman, J. Bielak, O. Ghattas, K. liu Ma, and D. R. Ohallaron, "Abstract from mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing."

[9] C. M. Hoffmann, *Implicit Curves and Surfaces in CAGD*, vol. 13. IEEE Institute of Electrical and Electronics, 1993.

[10] S. Forstmann, J. Ohya, A. Krohn-Grimberghe, and R. McDougall, "Deformation styles for spline-based skeletal animation," in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, p. 150, Eurographics Association, 2007.

[11] T. Sederberg and S. Parry, "Free-form deformation of solid geometric models," *ACM Siggraph Computer Graphics*, vol. 20, no. 4, pp. 151 160, 1986.

[12] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popovic, "Interactive skeleton-driven dynamic deformations," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 586 593, 2002.

[13] K. Zhou, J. Snyder, X. Liu, B. Guo, and H. Shum, "Large mesh deformation using the volumetric graph laplacian," June 22 2005. US Patent App. 11/158,428.

[14] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, p. 346, 2002.

[15] S. Schaefer and J. Warren, "Dual Marching Cubes: primal contouring of dual grids," in *Computer Graphics Forum*, vol. 24, pp. 195 201, Citeseer, 2005.

[16] W. Lorensen and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, p. 169, ACM, 1987.

[17] L. Kobbelt, M. Botsch, U. Schwanecke, and H. Seidel, "Feature sensitive surface extraction from volume data," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 57 66, ACM, 2001.

[18] R. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *ACM Siggraph Computer Graphics*, vol. 22, no. 4, p. 74, 1988.

[19] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, p. 458, ACM, 1994.

[20] M. Hadwiger, C. Berger, and H. Hauser, "High-quality two-level volume rendering of segmented data sets on consumer graphics hardware," in *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, p. 40, IEEE Computer Society, 2003.

[21] A. Kaufman, "Volume visualization," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, p. 167, 1996.

[22] J. Huang, K. Mueller, R. Crawfis, D. Bartz, and M. Meissner. "A practical evaluation of popular volume rendering algorithms," *Volume Visualization, 2000. VV 2000. IEEE Symposium on*, pp. 81 90, Oct. 2000.

[23] E. Gobbetti, F. Marton, and J. Iglesias Guitián, "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7, pp. 797–806, 2008.

[24] S. Lombeyda, M. Breen, and A. Heirich, "Scalable Interactive Ray-Casting of Volumes Using Off-the-Shelf Components," 2008.

[25] A. Requicha, H. Voelcker, and N. Y. P. A. P. Rochester Univ., "Constructive solid geometry," tech. rep., Rochester Univ., N. Y. Production Automation Project, 1977.

[26] J. Davy and P. Dew, "A polymorphic library for constructive solid geometry," *Journal of Functional Programming*, vol. 5, no. 03, pp. 415–442, 2008.

[27] B. Wyvill, A. Guy, and E. Galin, "Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system," in *Computer Graphics Forum*, vol. 18, pp. 149–158, John Wiley & Sons, 2001.

[28] C. Loop, "Smooth subdivision surfaces based on triangles," *Master's thesis, University of Utah, Department of Mathematics*, 1987.

[29] J. Warren and H. Weimer, *Subdivision methods for geometric design: A constructive approach*. Morgan Kaufmann Pub, 2001.

[30] G. Morin, J. Warren, and H. Weimer, "A subdivision scheme for surfaces of revolution," *Computer Aided Geometric Design*, vol. 18, no. 5, p. 483, 2001.

[31] S. Schaefer, J. Warren, and D. Zorin, "Lofting curve networks using subdivision surfaces," in *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, p. 114, ACM, 2004.

[32] J. Richards and X. Jia, *Remote sensing digital image analysis*. Springer Berlin etc., 1993.

[33] P. Lions, J. Morel, and T. Coll, "Image selective smoothing and edge detection by nonlinear diffusion," *SIAM Journal on Numerical Analysis*, vol. 29, no. 1, pp. 182 193, 1992.

[34] A. Buades, B. Coll, and J. Morel, "A non-local algorithm for image denoising," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005*, vol. 2, 2005.

[35] T. Ju, "Robust repair of polygonal models," in *ACM SIGGRAPH 2004 Papers*, p. 895, ACM, 2004.

[36] S. Bischoff, D. Pavic, and L. Kobbelt, "Automatic restoration of polygon models," *ACM Transactions on Graphics*, vol. 24, no. 4, pp. 1332-1352, 2005.

[37] B. Houston, M. Nielsen, C. Batty, O. Nilsson, and K. Museth, "Hierarchical RLE level set: A compact and versatile deformable surface representation," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 1, p. 175, 2006.

[38] A. Hornung and L. Kobbelt, "Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information," in *Proceedings of the fourth Eurographics symposium on Geometry processing*, p. 50, Eurographics Association, 2006.

[39] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and vision computing*, vol. 21, no. 11, pp. 977-1000, 2003.

[40] M. Jenkinson, P. Bannister, M. Brady, and S. Smith, "Improved optimization for the robust and accurate linear registration and motion correction of brain images," *Neuroimage*, vol. 17, no. 2, pp. 825–841, 2002.

[41] W. E. Lorensen and H. 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*, (New York, NY, USA), pp. 163-169, ACM, 1987.

[42] T. Ju, F. Losasso, S. Schaefer, and J. Warren, "Dual contouring of hermite data," in *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 339–346, ACM, 2002.

[43] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," in *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 65–74, ACM, 1988.

[44] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, 1990.

[45] D. Laur and P. Hanrahan, "Hierarchical splatting: a progressive refinement algorithm for volume rendering," *SIGGRAPH Comput. Graph.*, vol. 25, no. 4, pp. 285–288, 1991.

[46] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 451–458, ACM, 1994.

[47] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware," tech. rep., Chapel Hill, NC, USA, 1994.

[48] O. Wilson, A. VanGelder, and J. Wilhelms, "Direct volume rendering via 3d textures," tech. rep., Santa Cruz, CA, USA, 1994.

[49] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, (New York, NY, USA), pp. 91–98, ACM, 1994.

[50] R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," in *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 169–177, ACM, 1998.

[51] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization," in *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, (New York, NY, USA), pp. 109–118, ACM, 2000.

[52] K. Engel, M. Kraus, and T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, (New York, NY, USA), pp. 9–16, ACM, 2001.

[53] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution volume visualization with a teuxture-based octree," *The Visual Computer*, vol. 17, pp. 185–197,

May 2001.

[54] E. LaMar, B. Hamann, and K. I. Joy, "Multiresolution techniques for interactive texture-based volume visualization," in *VIS '99: Proceedings of the conference on Visualization '99*, (Los Alamitos, CA, USA), pp. 355-361, IEEE Computer Society Press, 1999.

[55] B. Wilson, K.-L. Ma, and P. S. McCormick, "A hardware-assisted hybrid rendering technique for interactive volume visualization," *Volume Visualization and Graphics, IEEE Symposium on*, vol. 0, pp. 123-130, 2002.

[56] C. A. Dietrich, L. P. Nedel, S. D. Olabarriaga, J. L. D. Comba, D. J. Zanchet, A. M. M. da Silva, and E. F. de Souza Montero, "Real-time interactive visualization and manipulation of the volumetric data using gpu-based methods," vol. 5367, pp. 181-192, SPIE, 2004.

[57] B. Naylor, J. Amanatides, and W. Thibault, "Merging bsp trees yields polyhedral set operations," in *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 115-124, ACM, 1990.

[58] A. Requicha and H. Voelcker, "Boolean operations in solid modeling: Boundary evaluation and merging algorithms," *Proceedings of the IEEE*, vol. 73, no. 1, pp. 30-44, 1985.

[59] D. Ayala, P. Brunet, R. Juan, and I. Navazo, "Object representation by means of nonminimal division quadtrees and octrees," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 1, p. 59, 1985.

[60] D. Laidlaw, W. Trumbore, and J. Hughes, "Constructive solid geometry for polyhedral objects," *ACM SIGGRAPH Computer Graphics*, vol. 20, no. 4, p. 170, 1986.

[61] C. M. Hoffmann, *Geometric and solid modeling: an introduction*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989.

[62] A. A. Requicha and H. B. Voelcker, "Boolean operations in solid modelling: Boundary evaluation and merging algorithms," 1984.

[63] A. Kaufman and E. Shimony, "3D scan-conversion algorithms for voxel-based graphics," in *Proceedings of the 1986 workshop on Interactive 3D graphics*, p. 75, ACM, 1987.

[64] S. Fang and R. Srinivasan, "Volumetric-CSG–a model-based volume visualization approach," in *Proc. 6th International Conference in Central Europe on Computer Graphics and Visualization*, pp. 88-95, Citeseer, 1998.

[65] E. Eisemann and X. Décoret, "Fast scene voxelization and applications," in *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, p. 78, ACM, 2006.

[66] E. Eisemann and X. Décoret, "Single-pass gpu solid voxelization for real-time applications," in *Proceedings of graphics interface 2008*, pp. 73–80, Canadian Information Processing Society, 2008.

[67] J. Georgii, J. Kruger, and R. Westermann, "Interactive gpu-based collision detection," 2007.

[68] K. B?rger, S. Hertel, J. Kr?ger, and R. Westermann, "Gpu rendering of secondary effects," 2007.

[69] J. Hable and J. Rossignac, "Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes," in *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, (New York, NY, USA), pp. 1024–1031, ACM, 2005.

[70] F. Romeiro, L. Velho, and L. H. de Figueiredo, "Scalable gpu rendering of csg models," *Computers and Graphics*, vol. 32, no. 5, pp. 526 539, 2008.

[71] F. Romeiro, L. Velho, and L. H. de Figueiredo, "Hardware-assisted csg rendering," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, (New York, NY, USA), p. 119, ACM, 2006.

[72] C. Everitt, "Interactive order-independent transparency," 2001.

[73] Y. Chen and C. Wang, "Layer Depth-Normal Images for Complex Geometries-Part One: Accurate Modeling and Adaptive Sampling," in *Proc. ASME Computers and Information in Engineering Conference, Brooklyn, NY*, 2008.

[74] D. Pavic, M. Campen, and L. Kobbelt, "Hybrid booleans," in *Computer Graphics Forum*, vol. 29, pp. 75–87, Blackwell Publishing, 2010.

[75] S. Bischoff, D. Pavic, and L. Kobbelt, "Automatic restoration of polygon models," *ACM Trans. Graph.*, vol. 24, no. 4, pp. 1332–1352, 2005.

[76] S. Roth, "Ray casting for modeling solids* 1," *Computer Graphics and Image Processing*, vol. 18, no. 2, pp. 109–144, 1982.

[77] F. Bernardon, C. Pagot, J. Comba, and C. Silva, "Gpu-based tiled ray casting using depth peeling," *Journal of Graphics, GPU, and Game Tools*, vol. 11,

pp. 1–16, 2006.

[78] T. Lokovic and E. Veach, "Deep shadow maps," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 385 392, ACM Press/Addison-Wesley Publishing Co., 2000.

[79] G. Kurka, "Depth-complexity based occluder selection," in *ACM SIGGRAPH 2002 conference abstracts and applications*, p. 165, ACM, 2002.

[80] V. Forest, L. Barthe, and M. Paulin, "Accurate shadows by depth complexity sampling," in *Computer Graphics Forum*, vol. 27, pp. 663–674, John Wiley & Sons, 2008.

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

[82] D. Smythe, "A two-pass mesh warping algorithm for object transformation and image interpolation," *Rapport technique*, vol. 1030, 1990.

[83] T. Ju, J. Warren, G. Eichele, C. Thaller, W. Chiu, and J. Carson, "A geometric database for gene expression data," in *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, p. 176, Eurographics Association, 2003.

[84] F. Bookstein, "Principal warps: Thin-plate splines and the decomposition of deformations," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 11, no. 6, pp. 567–585, 1989.

[85] T. Beier and S. Neely, "Feature-based image metamorphosis," *ACM SIGGRAPH Computer Graphics*, vol. 26, no. 2, pp. 35–42, 1992.

[86] R. MacCracken and K. Joy, "Free-form deformations with lattices of arbitrary topology," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 181–188, ACM, 1996.

[87] S. Lee, K. Chwa, and S. Shin, "Image metamorphosis using snakes and free-form deformations," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, p. 448, ACM, 1995.

[88] D. Shepard, "A two-dimensional interpolation function for irregularly-spaced data," in *Proceedings of the 1968 23rd ACM national conference*, pp. 517–524, ACM, 1968.

[89] K. Kobayashi and K. Ootsubo, "t-FFD: free-form deformation by using triangular mesh," in *Proceedings of the eighth ACM symposium on Solid modeling and applications*, p. 234, ACM, 2003.

[90] T. Igarashi, T. Moscovich, and J. Hughes, "As-rigid-as-possible shape manipulation," *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 1134-1141, 2005.

[91] M. Alexa, D. Cohen-Or, and D. Levin, "As-rigid-as-possible shape interpolation," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 157–164, ACM Press/Addison-Wesley Publishing Co., 2000.

[92] D. Levin, "The approximation power of moving least-squares," *Mathematics of computation*, vol. 67, no. 224, pp. 1517–1531, 1998.

[93] X. Gu and S. Yau, "Global conformal surface parameterization," in *Proceedings*

of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing, p. 137, Eurographics Association, 2003.

[94] B. Horn *et al.*, "Closed-form solution of absolute orientation using unit quaternions," *Journal of the Optical Society of America A*, vol. 4, no. 4, pp. 629-642, 1987.

[95] G. Wolberg, "Image morphing: a survey," *The Visual Computer*, vol. 14, no. 8, pp. 360–372, 1998.

[96] B. Tiddeman, N. Duffy, and G. Rabey, "A general method for overlap control in image warping," *Computers & Graphics*, vol. 25, no. 1, pp. 59-66, 2001.

[97] H. Lei and V. Govindaraju, "Direct image matching by dynamic warping," 2004.

[98] D. Keysers and W. Unger, "Elastic image matching is NP-complete," *Pattern Recognition Letters*, vol. 24, no. 1-3, pp. 445-453, 2003.

[99] S. Shen, J. Duan, J. Fiveash, I. Brezovich, B. Plant, S. Spencer, R. Popple, P. Pareek, and J. Bonner, "Validation of target volume and position in respiratory gated CT planning and treatment," *Medical Physics*, vol. 30, p. 3196, 2003.

[100] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl, "Level-of-detail volume rendering via 3D textures," in *Proceedings of the 2000 IEEE symposium on Volume visualization*, pp. 7-13, ACM, 2000.

[101] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive volume on standard PC graphics hardware using multi-textures and multi-stage

rasterization," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, p. 118, ACM, 2000.

[102] E. LaMar, B. Hamann, and K. Joy, "Multiresolution techniques for interactive texture-based volume visualization," in *Proceedings of the conference on Visualization'99: celebrating ten years*, pp. 355-361, IEEE Computer Society Press, 1999.

[103] N. Gagvani and D. Silver, "Animating volumetric models," *Graphical Models*, vol. 63, no. 6, pp. 443–458, 2001.

[104] V. Singh, D. Silver, and N. Cornea, "Real-time volume manipulation," in *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, p. 51, ACM, 2003.

[105] S. Fang, R. Srinivasan, R. Raghavan, and J. Richtsmeier, "Volume morphing and rendering-an integrated approach," *Computer Aided Geometric Design*, vol. 17, no. 1, pp. 59 81, 2000.

[106] T. He, S. Wang, and A. Kaufman, "Wavelet-based volume morphing," in *Proceedings of the conference on Visualization'94*, p. 92, IEEE Computer Society Press, 1994.

[107] S. Walton and M. Jones, "Interacting with Volume Data: Deformations using Forward Projection," in *International Conference on Medical Information Visualisation-BioMedical Visualisation, 2007. MediVis 2007*, pp. 48–53, 2007.

[108] B. Horn and B. Schunck, "Determining optical flow," *Artificial intelligence*, vol. 17, no. 1-3, pp. 185-203, 1981.

[109] E. Castillo, R. Castillo, Y. Zhang, and T. Guerrero, "Compressible image registration for thoracic computed tomography images," *Journal of Medical and Biological Engineering*, vol. 29, no. 5, 2009.

[110] S. Schaefer, T. McPhail, and J. Warren, "Image deformation using moving least squares," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, p. 540, 2006.

[111] T. Mcphail, P. Feng, and J. Warren, "Fast Cube Cutting for Interactive Volume Visualization," *Advances in Visual Computing*, pp. 620-631, 2009.

[112] T. Cormen, *Introduction to algorithms*. The MIT press, 2001.

[113] J. Carson, T. Ju, H. Lu, C. Thaller, M. Xu, S. Pallas, M. Crair, J. Warren, W. Chiu, and G. Eichele, "A digital atlas to characterize the mouse brain transcriptome," *PLoS Comput Biol*, vol. 1, no. 4, p. e41, 2005.

[114] A. Curtis, M. Powell, and J. Reid, "On the estimation of sparse Jacobian matrices," *J. Inst. Math. Appl*, vol. 13, no. 1, 1974.

[115] W. Li, K. Mueller, and A. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, (Washington, DC, USA), p. 42, IEEE Computer Society, 2003.