GPU-based volume deformation.

Walton, Simon

GPU-Based Volume Deformation

Simon Walton BSc. (Wales)

Submitted to the University of Wales in
fulfilment of the requirements for the Degree of
Doctor of Philosophy

Swansea University
Prifysgol Abertawe

October 2007

Summary

Surface-based representations of objects, and consequently their rendering algorithms, currently dominate the field of computer graphics. It could be argued that this is not just due to the efficiency of representation (representing merely surfaces, and not internal information), but is mostly due to the fact that surface-based graphics as a sub-field has seen many years of prioritised research and development. *Volume graphics* as a sub-field of computer graphics has however seen a rapid rise in research concentration in recent years. Its popularity can be attributed mainly to its ever-important role in medical applications such as surgery simulations and medical illustration; however, its rapid growth in the past five years or so is unquestionably due to the real-time volume rendering techniques implemented on the Graphics Processing Units of commodity graphics hardware.

The deformation of graphical objects is an important part of animation; particularly in CGI-based movies where characters must bend and stretch comically according to their actions. Deformation also plays an important role in surgical simulations, where real-time physically-based solutions are required to give the surgeon or student a realistic simulation of a surgical operation. The deformation of volumetric data (as in volume graphics) is a challenge due to the sheer amount of data that must be transformed, and the lack of topographical/semantic information that is embedded with freshly-aquired data. Such semantics must usually be inferred by the user using manual processes such as segmentation.

The work presented in this thesis provides a robust set of methods and techniques for the real-time manipulation of volumetric data, utilising high-performance graphics hardware to ensure that the field of volume graphics can continue to be a highly-attractive alternative to surface-based graphics. The main contributions of this work are:

- A comprehensive review of volume graphics and volume deformation;
- An introduction to important GPU-acclererated volume graphics methods;
- A framework for the non-reconstructive deformation of volume data;
- A GPU-accelerated forward-projection system for interactive volume deformation;
- A real-time backward-mapping raycasting renderer for interactive, character-based volume deformation.

Parts of this thesis have been published internationally in the Journal of the Winter School of Computer Graphics 2006 [WJ06], and the Fourth International Conference Medical Information Visualisation - BioMedical Visualisation 2007 [WJ07].

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed .... (candidate)

Date ....21/01/08........

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed .... (candidate)

Date ....21/01/08........

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date ....21/01/08........

Acknowledgements

I find it rather difficult to believe that I have reached the end of my PhD at Swansea; it must certainly be true that time flies when you're having fun. Undertaking a PhD has been the most challenging chapter of my life to date, but one I have thoroughly enjoyed.

My first acknowledgement goes to my parents for offering their support for me to undertake a PhD. I would also like to thank my supervisor, Dr Mark Jones for his seemingly infinite supply of optimism when I would visit him with nothing but pessimism about my work; Dr Jones would always convince me of the existence of that silver lining. My thanks also go to my secondary supervisor, Prof. Min Chen, for his invaluable advice on my direction.

I would finally like to thank all of the wonderful people I have worked with in Swansea over the past three years. I'd particularly like to thank Chris Miller for his advice on GPU development, and the very long loan of books for this period.

Contents

Chapter 1

Introduction

Contents

The field of computer graphics has seen an incredible growth in popularity in the past 20 years due to its applications in the entertainment industry. Movies such as Tron, although primitive by today's standards, showed the potential for the use of computer generated imagery in the media. Today, it is difficult to find a film that does not take advantage of 3D rendered scenes; for some films such as Star Wars, almost the entire set is virtual. Companies such as Pixar and Dreamworks have additionally provided a boost to the industry by producing award-winning pure CGI films such as Toy Story and Shrek.

It is clear that the majority of computer imagery generated from 3D models is produced from surface representations, since they provide a compact representation of an object with the availability of the large number of surface-based rendering algorithms that have been developed and refined over the years.

An increasingly available alternative however is the field of *volume graphics*, which has established itself as an important area of research in the computer graphics community; mainly due to its prominent usage in medical visualisation. With the advent of vast improvements to CT and MRI scanning, doctors are increasingly looking to the introduction of advanced 3D visualisation techniques to view the datasets.

1.1 Aims and Objectives

The majority of literature on volume graphics is concerned mainly with the visualisation aspect – that is, the synthesis of 2D images from the 3D discrete data. Early work on volume rendering was conducted by researchers such as Levoy [Lev88], Kaufman[ASK94], and Westover [Wes89], all of which have established the fundamentals of volume rendering approaches; and also the field of volume graphics as its own area of research [KCY93].

Volume graphics, in comparison with surface-based graphics, suffers from a lack of research into data manipulation techniques; in particular, the amount of research into interactive manipulation is low in comparison to surface-based manipulation techniques. Tools available for manipulation sometimes resort to unintuitive user interfaces that expose the low-level data structure to the user. The user should not be concerned with such detail, and should be offered a higher level of abstraction. In addition, many approaches necessitate specialised rendering algorithms or convert to an intermediate representation such as a surface mesh. This however restricts the possibility of implementation on graphics hardware, and often removes the main benefit of direct volume rendering – the availability of complete internal texture information rather than some approximation of a surface using isosurface generation algorithms.

The most recent works in the area of volume rendering are mainly concerned with GPU implementations of the well-established raycasting algorithms. Two factors drive this research: firstly, the availability and price of modern consumer graphics hardware, and secondly, the need for interactive rendering of volume datasets. Consumer graphics hardware units are most typically used for a very fixed pipeline of rasterising triangular meshes, with programmable units for manipulating the vertices and shading the resulting pixels. However, as the hardware technology advances, the programmability and the ease of development of such cards increases. The result of this increase in power and programmability is that modern graphics hardware units are used for purposes other than what they were originally intended to; for example, the implementation of raytracing algorithms, and even general purpose computation.

The objectives for the work contained in this thesis are:

1. To develop a methodology and framework for the deformation of volumetric datasets that will enable users to produce global deformations in an intuitive manner. The framework must support the use of traditional raycasting algorithms.
2. To investigate the feasibility of forward and backward mapping approaches for volume deformation in real-time.
3. To study visual enhancements to volume rendering through the use of global illumination techniques.
4. To provide a complete tool for intuitive and interactive volume deformation.

1.2 Thesis Outline

The work contained within this thesis is divided into two components, namely the *review chapters* and the *research chapters*. The former components may be found in the following two chapters (Chapters 2 & 3) where a review of the field of volume graphics is conducted, followed by a chapter on the deformation and animation of volumetric data. These chapters set the foundations for the thesis, and give the reader a good working knowledge of the area before the research contributions of the thesis are given.

These review chapters are followed by the research chapters, which provide the contributions made by this thesis to the field of volume graphics. Chapter 4 can be considered as a transition chapter, and provides firstly an extensive review of GPU volume rendering techniques including example shader implementations, and secondly a GPU isosurface intersection framework for ray tracing applications. The remaining chapters introduce the Volume Wires volume deformation framework and methodology and give GPU-based implementations, finishing with the implementation of a complete tool for volume deformation.

The below gives a synopsis of each following chapter contained within this thesis.

1.2.1 Chapter 2: Volume Graphics

This chapter gives the reader an introduction to the subfield of computer graphics known as *volume graphics*. It begins with a discussion of the fundamentals of the subject, including a review of the most important methods for acquiring volumetric data. The reader is then introduced to a variety of modeling schemes used in computer graphics, including volume modeling as an alternative modeling scheme. The visualisation of volumetric data follows, and details the various rendering algorithms that currently exist for rendering volumetric data to the screen, including a review of important acceleration techniques that can be implemented to improve rendering times. Distance fields, an important concept utilised later in the thesis, are next introduced; followed finally by a review of filtering methods for volumetric data.

1.2.2 Chapter 3: Deformation

A technique for the deformation of volumetric data is one of the major contributions of this thesis. This chapter first gives the reader an introduction to deformation and animation techniques in general, as used mostly in the entertainment industry with surface-based models. The deformation of volumetric data is then introduced as a field of research, including a discussion on the surrounding issues of deforming discretely sampled data such as volume dataset, in comparison with the deformation of surface meshes. A comprehensive review is then given of volume deformation techniques that currently exist, using a variety of methodologies (from the perspective of the user) and rendering algorithms. The segmentation of volumetric data is often an important step in providing the necessary semantic information for deformation and animation, and therefore a review of segmentation methods for volumetric data is given to conclude the chapter.

1.2.3 Chapter 4: GPU Volume Rendering

A comprehensive review of GPU-based volume rendering techniques is first given in this chapter. This includes an overview of the OpenGL pipeline (OpenGL is the graphics API used throughout this thesis), a review of the current state-of-the-art research into volume rendering on GPUs, and low-level details of an example GPU-based raycasting volume renderer. The latter half of the chapter introduces a method for providing GPU-accelerated volume isosurface intersections to a standard ray tracing pipeline, allowing the CPU to concentrate on intersections with surface-based objects, and the GPU intersections with the volume isosurface of choice. In addition, we show the advantage of leveraging the power of the GPU when deformations of the volume dataset are introduced in the form of Spatial Transfer Functions. Full global illumination of the isosurface is provided by the ray tracer; however, the ray tracer is not the focus of the research.

1.2.4 Chapter 5: Volume Wires

In this chapter, a volume deformation methodology and framework is presented, called *Volume Wires*. The methodology is based on curve-skeletons defined with the volume dataset, which are then manipulated to (in turn) manipulate the volume dataset. Because the method can be viewed as a volume sweeping technique, the chapter first gives a brief introduction to the related work on sweep representations in computer graphics. A high-level overview of deforming with the Volume Wires methodology is next given, including the effects that can be achieved by wire manipulation. This is followed by low-level details on the required mapping between object and world space, including details on how a one-way version of this mapping can be effectively encoded into a volume dataset. To demonstrate the framework's effectiveness, a raycasting rendering algorithm for the framework is finally given.

1.2.5 Chapter 6: Forward-Projection of Volume Wires Mapping

The previous chapter introduced the Volume Wires methodology and framework to the reader, and additionally provides an effective mechanism to encode the mapping. Although the discussed renderer provides good results, it is far from interactive as it requires the computation of the mapping encoding beforehand. This chapter therefore introduces a forward-projection implementation of the Volume Wires framework that provides real-time manipulation functionality to the user. Additionally, an analysis of the problem of image-space cracking artefacts is discussed, and a feasible solution for the system introduced. The forward-projection rendering algorithm and mapping function are evaluated on the GPU for optimal performance.

1.2.6 Chapter 7: A Complete Volume Deformation Tool

This chapter takes the Volume Wires framework into its final conclusion, providing a real-time raycasted volume deformation tool. The tool incorporates segmentation functionality based on energy-minimising splines, allowing the user to associate subvolumes within the dataset with each wire defined in the scene. This allows for effective character-based deformation without first resorting to manually labelling each voxel. The rendering algorithm employed utilises the segmentation data to generate ray entry/exit points, and provides a real-time raycasting of the deformed volume data evaluated on the GPU without any reconstruction.

1.3 Terminology

This thesis uses the consistent terminology given in Table 1.1 throughout. Where deviations from such terminology occur, the text will make such deviations clear.

Symbol	Description
$\mathbf{V}$	Volume dataset
$\mathbf{V}(x, y, z)$	Result of sampling volume dataset at position $(x, y, z) \in \mathbb{E}^3$
$\mathbb{E}^3$	3D Euclidean space
$\mathbb{R}$	Set of all real numbers
$\mathbb{N}$	Set of all natural numbers
ϵ	An arbitrarily small amount
t	A parametric offset
τ	An isosurface threshold
Φ / Φ^{-1}	Forward / Backward spatial transfer function

Table 1.1: Thesis Nomenclature

1.4 Volume Datasets

The volume datasets used for this thesis have been obtained from a variety of sources, and are all publicly available. The author acknowledges Stefan Roettgar's Volume Library [Roe] as an extremely valuable centralised source for such datasets. Some datasets used, such as the Visible Human torso dataset, are customised versions of these datasets.

Dataset	Acknowledgement
CT Head	University of North Carolina
CT Foot	Philips Research, Hamburg, Germany
CT Knee	Brigham and Women's Hospital Surgical Planning Laboratory
CT Carp	Michael Scheuring, University of Erlangen, Germany
Visible Human	National Library of Medicine
Lobster	VolVis distribution of SUNY Stony Brook
Bunny	Terry Yoo, National Library of Medicine
Tooth	GE Aircraft Engines, Evendale, Ohio, USA

Table 1.2: Volume datasets used in this thesis

Chapter 2

Volume Graphics

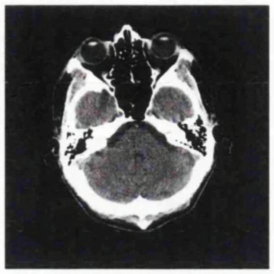
Contents

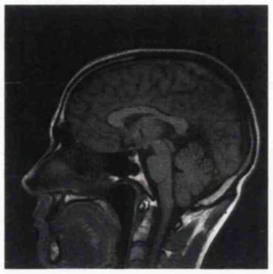
The field of *Volume Graphics* [KCY93] deals with the modelling and visualisation of *volume data*. A *volume* is defined as three-dimensional discretely sampled object – the most useful analogy is that of a regular two-dimensional image extended into the third dimension. The samples in a volume dataset can be obtained from sampling a real-life object, or they can be synthetically produced. *Volume data* is most commonly obtained by scanning real-world data, in a similar manner to a digital camera's CCD sensor sampling the light entering through the lens in a regular grid.

The sample values at each point in the three-dimensional grid are referred to as *voxels* (a word derived from the words *volume element*), as opposed to *pixels* in the two-dimensional case. The voxels typically represent some measure of density or opacity. For example, the output of a CT scanner is a volume dataset with values at each voxel representing the density of material found at that point.

Initial work in the field was conducted mainly on the visualisation of data obtained from medical imaging devices [Lev88]; the idea that representing three-dimensional objects discretely could be a rival representation to to the surface mesh was not considered. It soon became clear however that representing objects in this way provided a neat graphics framework, an idea proposed by Kaufman *et al.* [KCY93] in 1993.

This chapter gives an introduction to volume graphics as a field of computer graphics, including reviews of some of the most important techniques for visualising, modelling, and manipulating volume datasets.


(a) A slice from a CT scan *Image Credit:* [Wika]


(b) A slice from an MRI scan *Image Credit:* [Wikb]

(c) Ultrasound *Image Credit:* [Wikc]

Note: all images in this thesis are the creation of the author, except where explicit acknowledgement is given otherwise.

2.1 Data Acquisition

The acquisition of volume data is most commonly obtained from scanning a physical object to determine its internal structure. This section looks at three popular data acquisition methods, all of which have their primary use in medicine; and additionally looks at the acquisition of data from scientific simulations.

2.1.1 CT Scanning

Computed Tomography (commonly abbreviated to CT) is a process of medical image acquisition that requires patients to be scanned in a slice-by-slice manner. It was devised in 1972 by Godfrey Hounsfield [Hou73], who was awarded the Nobel prize in medicine jointly with Allan Cormack (who independently invented the method). Figure 2.1(a) shows a slice from a CT scan of a patient's head.

In order to perform a CT scan of a human patient, the patient is typically placed in a lying down position and the machine is lined up with the current slice of the patient to be scanned. X-rays are now fired into the patient which are in turn picked up by detectors positioned opposite the source. The equipment rotates around the current slice to complete the acquisition, and the data is constructed into a 2D image by a process termed *tomographic reconstruction*.

The values computed at each point represent the density of the material encountered by the x-rays, and these values are measured in Hounsfield units (after Godfrey Hounsfield). Values of around 400 represent dense objects such as bone, and values of around 100 represent soft tissue, skin, and some muscle. Values less than 100 will either represent very sparse internal material such as fat/water, or the air surrounding the patient. These values are usually mapped to 12-bit unsigned numbers, giving a range of 4096. Because the Hounsfield value for air is around -1000, these mapped values are shifted by 1000 to confirm to the unsigned data type.

A criticism of CT is that it is not well-suited to scanning soft tissues, and often produces quite low contrast in these soft-tissue value ranges. To partially improve the contrast in such areas, the patient is often given a contrast-enhancing agent (the most common agents used are Iodine and Barium) either in injection or pill form. Intravenous methods are used to highlight blood vessels and enhance contrast in areas carrying a large amount of blood – e.g. the brain or kidneys; whereas the pill form is most utilised for enhancing the digestive system. CT scanning is therefore best suited to tissues with a relatively high atomic density such as bone, and is the scanning method of choice for detecting abnormalities with these hard structures, such as fractures.

Early CT scanners typically offered a much lower resolution and a higher scanning time than the scanners that exist today. Over time, the understanding of the technology and reconstruction software improved. When this is coupled with higher performance computers, higher resolutions and lower scanning times resulted. Recent advancements in CT scanning by Philips have seen the introduction of their own Brilliance range of CT scanners, which are capable of performing full-body scans in less than 50 seconds. The scanner additionally offers slice resolution up to 1024^2 [Phi]. CT scanning has become so widespread and easily accessible that organisations such as Lifescan [Lif] offer 'peace of mind' CT scans for early diagnosis of many potentially serious health issues.

2.1.2 Magnetic Resonance Imaging (MRI)

Magnetic Resonance Imaging (MRI) was devised by Professor Raymond Vahan Damadian [Dam71] in 1971 as a method of detecting cancerous tissue. The foundations for his work were laid by Felix Bloch and Edward Purcell almost twenty years previously, both of whom received the Nobel Prize in Physics for their work on nuclear magnetic resonance and nuclear induction. Figure 2.1(b) shows a slice from an MRI scan of a patient's head; it is interesting to note the increased detail in the soft tissues in comparison with the CT slice from (a).

The MRI scanning procedure employs radio and magnetic waves to obtain information on the magnetic properties of the atoms inside the patient, rather than exposing the patient to potentially harmful radiation. The patient is exposed to strong magnetic fields, typically of around 0.5 to 3 Tesla but occasionally up to 7 Tesla; a higher strength of magnetic field gives a higher signal-to-noise ratio, at the expense of potentially harming the patient. These fields are generated by large magnets which are usually the main expense of an MRI scanner as they often have a high running cost due to their strength; indeed it is the case that MRI scanning equipment is many times more expensive to purchase and operate than CT scanning equipment.

During exposure to these fields, the hydrogen protons inside the patient's tissues align with the magnetic field and oscillate, a process known as the magnetic resonance phenomenon. A radio frequency pulse is applied through a set of coils, which pushes the aligned atoms away from the magnetic field. When the pulses stop, the protons revert to their original alignment. This in turn causes them to transmit energy that is picked up by another set of coils. The amount of energy detected is an indication of the density of water in the material, since dense materials have a higher density of hydrogen. It is for this reason that bones show as darker, less detailed objects, since bone does not contain as much hydrogen.

MRI is particularly well-suited to scanning soft tissues (such as the brain) due to the manner in which the object is scanned; soft tissues contain hydrogen atoms which are picked up extremely well by an MRI scan. MRI scanning, as with CT scanning, can be enhanced with contrast-enhancing agents that further improve the patient's response to the magnetic fields. The contrast in tissues with a higher atomic number such as bone is rather low, and for this reason CT scanning is the preferred method where detail is required in bone tissue.

2.1.3 Ultrasound

Ultrasound scanning was first used for medical imaging purposes in the late 1940's by Dr George Ludwig in Maryland, USA [obg]. Early experiments with the method from 1947 to 1949 demonstrated the effectiveness of using sound waves in the ultrasound range for detecting gall bladder stones. Ultrasound scanning has since proven so popular that the word 'ultrasound' is now used as a verb by the general public – meaning to perform an ultrasound scan. A large part of this is due to its popularity in showing real-time images of foetuses; it provides an important role in determining any physical abnormalities before the baby is born, and additionally, parents enjoy the experience of seeing their unborn child in real-time.

Ultrasound scanning is by far most accessible scanning method discussed in this section. The vast majority of scanners are handheld, and are applied to the region to be scanned. The skin is first prepared with a special gel which enables a greater contact between scanner and body, and also provides a smoother surface on which to slide the scanner. Ultrasonic sound waves are passed into the body, which are then received back by the device. Determining the physical structure of the objects detected is based on the strength of the returning echo, its direction, and the length of time it took to return. Images can be reconstructed in real-time via this simple process, which is one of the reasons ultrasound remains a popular scanning method.

Though ultrasound is a popular method for acquiring images from within the body, the method has only very recently provided volumetric data by employing larger-scale and more complex acquisition devices such as GE's LOGIQ 9 [GEL]. A recent development in the field of ultrasound scanning is that of 3D ultrasound scanning [ult], which directs multiple ultrasound waves from different locations to reconstruct a 3D model of the object found by the reflected waves. This can be taken a step further by introducing time as an additional variable, giving 4D ultrasound – a dynamically updating 3D model of the scanned object (see Figure 2.1). In addition, by analysing the Doppler shift of the returned echo (the compression and expansion of the waves), the rate of movement of the object can be inferred. Using this information, the rate of blood flow can be measured in real-time to diagnose any blood flow problems with the unborn child.

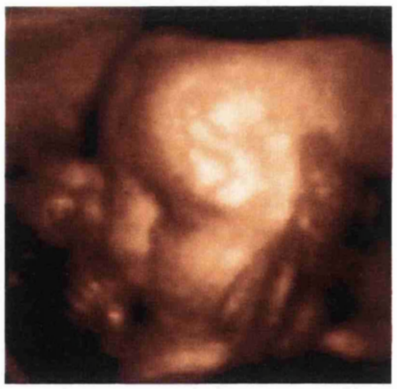


Figure 2.1: A 3D Ultrasound model of a baby's head
Image Credit: [Wikc]

2.1.4 Scientific Simulations

In the field of Computational Fluid Dynamics (CFD), simulations of fluids are commonly calculated using numerical methods such as the Navier-Stokes equations for fluid flow. A volume dataset is initialised with default values, and the fluid flow is calculated at each voxel for a particular time t by solving the equations. Volume rendering techniques can be used to render the resulting data by interpreting the data either as static or dynamic. The resulting datasets that are interpreted as static can be rendered quite simply using a standard direct volume rendering algorithm for giving a realistic interpretation of the liquid [SS91] or gaseous [EYSK94] phenomena.

The end result of a CFD simulation can often be a vector field, where each sample is represented as a vector denoting the direction of flow at that point. Such datasets are best rendered primarily using a forward-projection technique to give important direction cues such as arrows embedded in the dataset at areas of high flow.

2.2 Volume Modeling

There are many different data structures and representations for 3D objects stored in a computer's memory. A *model* refers to the abstraction, representation, and implementation of the object or phenomena being modeled [Bar92].

The choice of object representation depends on a number of factors:

- The manner in which the object is acquired or created – e.g. via scientific simulation, a CT scan, a laser scanning model, 3D surface-based modelling tool;
- The choice of rendering algorithm – since the input to a rendering algorithm is the graphical model in some format;
- The application – e.g. CAD/CAM packages use surface-based representations.

This section explores a variety of object representations used in computer graphics today.

2.2.1 Solid Modelling Schemes

Boundary Representation

Boundary representations represent only the boundary or *surface* of the object being modelled [FvDFH96, Req80]. Such a representation is defined in terms of vertices, connected via edges. These vertices are linked topologically to form primitives – the most common surface-based primitive in use today is a triangle. Triangles are chosen for their attractive mathematical properties which make them efficient candidates for surface-based rendering algorithms. Modern graphics cards are highly optimised for dealing with triangle primitives, and higher-level primitives specified by the programmer are usually broken down into triangles before sending to the graphics card.

Spatial Partitioning

Spatial partitioning methods attempt to represent the target object as a set of decomposed primitives. *Spatial Occupancy Enumeration* represents solid objects in terms of a fixed regular grid [FvDFH96], composed of primitive cells. Each cell is marked to denote whether the object is present at the cell or not absent. Such a binary scheme suffers from poor quality object reconstruction as there is no notion of partial occupancy in a cell – cells are either marked as present or absent.

Constructive Solid Geometry

Constructive Solid Geometry (CSG) [FvDFH96] uses boolean set operations to on objects to form new objects. Though such a representation has an inherently mathematical basis, it is an analogue of the real-world process of building solid objects, in which large objects are composed of many smaller objects put together in different ways. The object representation is a binary tree with the new object at the root; each node represents a Boolean operation (e.g. union $\cup$, intersection $\cap$, and difference $-$), and the object resulting from the operation with the two children. Obtaining the object at the root of the tree typically involves a depth-first traversal of the tree, evaluating the operations at each node until the final object is obtained.

2.2.2 Volume Modeling

A volume data type is a set of scalar fields that define some property for every point $p \in \mathbb{E}^3$. A scalar field is defined as a function $F : \mathbb{E}^3 \rightarrow \mathbb{R}$, and many of these fields can be combined to form a representation of a volume object, modeling the various properties at each point. The most common property modeled for volumetric representations is some measure of the density of opacity at each point.

The representation of a scalar field depends on the application. Chen *et al.* identify three approaches for the representation of a scalar field [CWRT02]:

- *Mathematically* – the scalar field definition function $F : \mathbb{E}^3 \rightarrow \mathbb{R}$ can be used to mathematically define the properties.
- *Procedurally* – a function or procedure can be defined in a programming language to calculate the property of each point $p \in \mathbb{E}^3$. Procedural scalar fields can therefore be considerably more complex than a mathematical definition.
- *Discretely* – the volume data type can be discretised into a finite set of voxels – a *volume dataset*. Associated with the volume dataset is an *interpolation function* to recover a continuous approximation to the original signal. This representation is most common output of medical imaging scanning equipment.

The overall modelling approach is termed *field-based modeling and rendering*, in which objects are modeled in terms of their scalar fields, with each scalar field representing a different attribute of the object, e.g. the red, green and blue colour components, opacity, and also more intricate details such as reflection/refraction coefficients. A *spatial object* is defined as one or more of these scalar fields combined as $o = \{O, A_1, \ldots, A_n\}$. The most common attribute field is the opacity field since it is the most common interpretation of volumetric objects scanned from real-world sources – higher density values can be mapped to higher opacity values, under the assumption that denser objects are more opaque.

Volume Scene Graphs

Volume Scene Graphs were introduced by Nadeau [Nad00] as a solution to the problem of combining multiple volumetric objects in one scene. In a volume scene graph, the scene is represented as a tree structure, with the volume datasets contained within the leaves, and operators at the intermediate nodes. Where volumetric objects overlap, the operator applied to each point within the overlap region can be found in the parent node of the volume objects; a simple 'union' operator might be to simply take the largest value sampled. Volume scene graphs are heavily utilised in the volume graphics API *vLib* developed by Winter [WC01].

Constructive Volume Geometry (CVG)

The Constructive Solid Geometry representation has been extended to the volume domain by Chen and Tucker [CT00] to become Constructive Volume Geometry. CVG defines an algebraic framework for combining multiple volumetric objects using the Boolean operations defined in the regular CSG framework. CVG also allows for greater flexibility in the construction of procedural volume objects.

CVG uses the notion of spatial objects to define Boolean operators which operate on the scalar fields that comprise the objects. Algebraic signatures are used to define classes of spatial objects to ensure interoperability, and each class defines the operators that are valid for that class of spatial objects.

2.2.3 Volume Grid Types

The samples in a discrete volume dataset are most commonly stored as a 3D regular grid of values. The memory addressing for such a scheme is simple; given the dimensions d_x, d_y, d_z of a set of voxels of size $d_x d_y d_z$ at memory location *ptr*, a voxel at integer position $(x, y, z) \in \mathbb{N}^3$ is referenced as $ptr + z * (d_x d_y) + (d_x y) + x$.

Depending on the nature of the data acquisition however, the grid may not be as inherently simple as this. Volume data can be structured or unstructured, regular or irregular:

- *Structured* grids have implicit geometry – that is, there is no need for the specification of the position of each sample and its connectivity to other samples; the position is implicitly inferred from the structure of the dataset. *Unstructured* grids therefore require explicit specification of the sample positions and their connectivity, since this information cannot be automatically inferred.

- *Regular* grids contain cells which all exhibit the same geometrical structure – that is, the geometry and neighbour count of each cell is the same. *Irregular* grids can differ in their geometrical shape and structure from sample to sample, making recovery of the samples substantially more complex.

Figure 2.2 demonstrates the four grid types.

A common approach for dealing with unstructured data is to revoxelise it into a regular, structured volume dataset which can then be rendered using the standard volume rendering algorithms; this approach however presents a trade-off between the quality of the final reconstruction and the size of the new volume dataset.

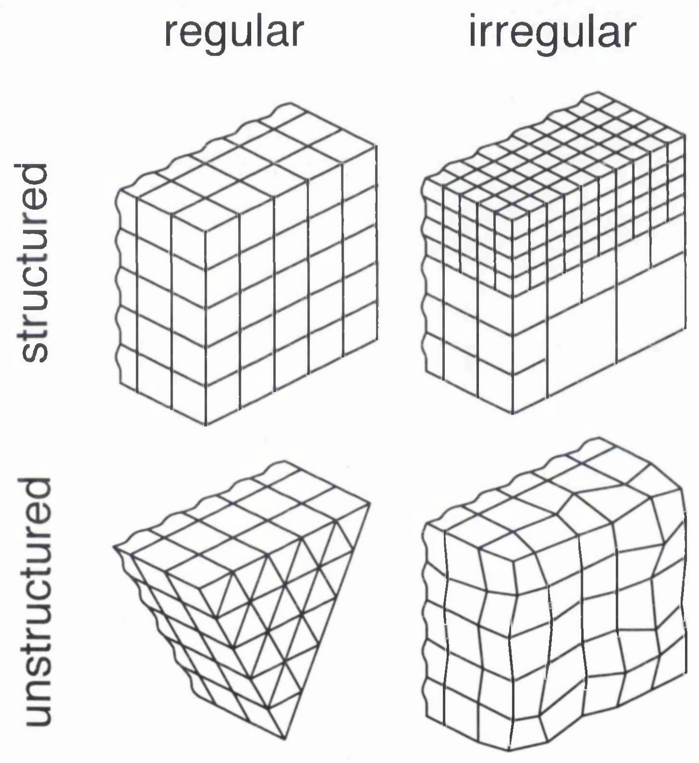


Figure 2.2: Grid Types

2.3 Volume Visualisation

Volume visualisation methods produce a two-dimensional image from volumetric data in a meaningful manner. Because of the availability of internal information (as opposed to just an infinitely thin surface), careful consideration is required as to how this information is displayed in the final image. Volume rendering algorithms can be classified as either *indirect* or *direct*. *Indirect* volume rendering algorithms first convert the volume data to an intermediate (perhaps surface-based) representation, and then use an appropriate rendering algorithm for this intermediate representation; whereas *direct* volume rendering algorithms render the volume data directly from the volumetric representation.

In the area of surface graphics, there exist both forward and backward projection algorithms. Forward-projection algorithms project the data from world space to image-space (the screen). Forward-projection algorithms for surface graphics are typically utilised in 3D games, with the bulk of the computation being performed by dedicated graphics hardware that is heavily optimised for rasterising triangular meshes. Pixar's RenderMan software is a forward-projection renderer primarily, but it also has ray tracing capabilities for additional effects such as complex reflections.

Backward-projection algorithms work the other way around, building an image by obtaining data for each pixel that comprises the image. The most common algorithms in this class are referred to as ray casters, in which imaginary rays of light are projected either from the world to the image or from the image into the world. Whatever data is encountered along the path of the ray is the data that will be used to shade the pixel.

In general, a volume dataset is visualised by using a discretised implementation of the natural behaviour of light as it travels through a semitransparent object to the eye – opaque objects near the viewer have the effect of blocking the light behind. The *volume rendering integral* determines how this light behaviour is modeled. There will be multiple pieces of information for each pixel in the final image, and these must be integrated together to form one final pixel colour. This process in general is called *compositing*, and a discussion on compositing methods can be found in the coming sections.

Because volume datasets often contain simple scalar values, colours and opacities must be mapped to these values in order to give a better visual representation. For example, it may be useful to assign a yellow-white colour to values in a medical dataset determined to be bone density, and a fleshy colour to lesser densities, assuming the these densities are hard and soft tissue. The functions that map voxel scalar values to colours and opacities are called *transfer functions*. In the coming text, $C(x)$ gives an RGB colour for a scalar value $x \in [0, 1]$, and $\alpha(x)$ gives the opacity, where the opacity is in $[0, 1]$. If available, a 3D RGB texture can be used to look up the colour values. This full-colour information is available for the Visible Human dataset, where the body was cut into extremely thin slices, and a photograph taken of each slice.

Before discussing direct volume rendering, an overview of the process for obtaining alternative data representations from a volume dataset are given in the next section.

2.3.1 Isosurfaces and Indirect Rendering

Because of the large amount of data contained within most volume datasets, early volume rendering algorithms first converted the discrete data into an intermediate representation in an attempt to use a more efficient surface-based rendering algorithm; this process is often termed *indirect* volume rendering. This indirect rendering is often achieved by extracting one or more *isosurfaces* from the volume and rendering just these isosurfaces.

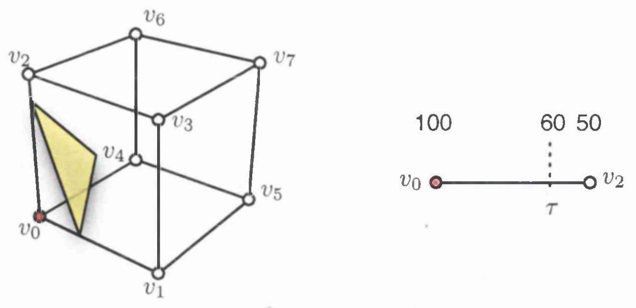


Figure 2.3: The Marching Cubes Algorithm

An isosurface $iso_{\mathbf{V}}$ within a volume dataset $\mathbf{V}$ is defined as:

$$iso_{\mathbf{V}}(\tau) = \{p \in \mathbf{E}^3 : \mathbf{V}(p) = \tau\} \tag{2.1}$$

that is, the set of points p equal to the specified value τ. Using an interpolation technique, the position of the isosurface for each point in each voxel cube can be calculated, if it exists. Whether the isosurface exists within a voxel cube can be calculated very efficiently by representing each cube (bounded by eight voxels) as an 8-bit byte b, with one bit representing each voxel as the voxel can be in one of two states: *above / equal* τ or *below* τ. Each bit is set to 1 if its voxel value is $\geq \tau$, and if the final byte unsigned value satisfies $0 < b < 255$, then the isosurface exists within the cube. Voxel cubes containing an isosurface are said to be *transverse*.

Given the computational complexity of compositing operations and the storage requirements of volume data, it is easy to see why indirect approaches can be favourable in some situations. By converting to a surface representation however, the internal texture information of the volume dataset is lost, and the renderer is simply left with an infinitely thin surface representation that gives only an estimation of one particular isosurface.

The next sections discuss the most popular algorithms for extracting isosurfaces from volume datasets, namely the *Marching Cubes* and *Marching Tetrahedra* algorithms.

Marching Cubes

The Marching Cubes algorithm [LC87] by Lorensen and Cline is the most well-known surface reconstruction algorithm, and calculates a list of surface primitives for a given volume dataset and isosurface value. For each transverse cube in the dataset, a surface primitive is computed representing the isosurface cutting through it (see Figure 2.3). This is achieved by comparing the transverse bitmask b against a precomputed table containing all combinations of primitive cutting through the cube. The total number of primitive configurations for each cube is 256 (2^8 for a cube bounded by 8 voxels), though the number of configurations can be reduced to 14 by observing that most of the configurations are simply rotated or reflected versions of others. Once a primitive list for a given transverse cube is established, the exact point of intersection with the cube edges is computed by linearly interpolating the two voxels connected to the edge (as shown in Figure 2.3).

A criticism of the Marching Cubes algorithm is that the method of identifying ambiguities is known to introduce holes into the resulting model, though the problem can be partially overcome by caching the edge intercepts for each face instead of recalculating for every connected cube. Another criticism is the often large number of triangles generated – severely impacting both the time of generation and the interactivity of the final mesh when rendered using a triangle rasterisation rendering algorithm. Schroeder *et al.* discuss methods for reducing the number of triangles [SZL92] by iterating over the resulting triangles and selecting vertex candidates for removal such that the resulting mesh is still a good approximation of the original.

Marching Tetrahedra

Payne and Toga [PT90] further developed the marching cubes algorithm in an attempt to avoid the ambiguous cases of Marching Cubes. Their method decomposes each voxel cube into up to five tetrahedra, making note of the tetrahedra intersected by the isosurface. The number of possible configurations can be simplified greatly from 16 to just three configurations when taking into account rotation and symmetrical equivalence. However, because of this additional decomposition step, the algorithm generates on average more primitives than Marching Cubes, adding to storage requirements and rendering times.

Treece *et al.* [TPG99] later develop the Regularised Marching Tetrahedra algorithm, which uses a vertex clustering scheme to achieve a triangle reduction rate of around 70% over standard Marching Tetrahedra.

2.3.2 Signal Reconstruction

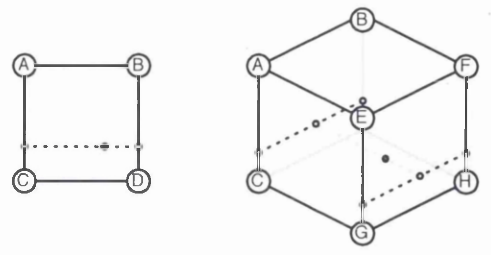
Most direct volume rendering algorithms, as will be discussed in coming sections, require access to the data 'between' voxels; the volume dataset should be treated as a continuous object rather than a discrete one. The process in general is called *interpolation* from a computer science or mathematical perspective, or *signal reconstruction* from an engineering perpective.

This section looks at three signal reconstruction techniques – namely *nearest-neighbour*, *trilinear*, and *tricubic*. The choice of technique is a trade-off between reconstruction quality and the computational complexity.

Zeroth-Order : Nearest-Neighbour Interpolation

Nearest-neighbour interpolation is the simplest method of all conceptually, is quick to compute, but produces very poor results. Nearest-neighbour interpolation chooses the nearest voxel to the current sample point within the dataset, obtained simply by adding 0.5 to the coordinate and taking the floor of the result:

$$\mathrm{V}_{nn}(x, y, z) = I(\lfloor x + 0.5 \rfloor, \lfloor y + 0.5 \rfloor, \lfloor z + 0.5 \rfloor) \tag{2.2}$$



(a) Bilinear (b) Trilinear

Figure 2.4: Bilinear Interpolation on a 2D area, and trilinear interpolation inside a 3D cube

This interpolation method produces very blocky, aliased results and should only be used when speed is of extreme importance and visual quality is of low importance. Its application therefore is mainly in interaction-intensive volume renderers.

First-Order : Trilinear Interpolation

In 1D, *linear interpolation* attempts to obtain a value that lies at position t inbetween two samples a and b as $a+((b-a)*t)$. An extension of this is *bilinear interpolation* in 2D, which computes first the linear interpolation of two edges of the cube, and then linearly interpolates these intermediate results for the final coordinate (see Figure 2.4(a)). *Trilinear interpolation* further extends the linear interpolation into three dimensions, by linearly interpolating the result of bilinear interpolation of two opposite faces of the voxel cube (Figure 2.4(b)). The ability to decompose the 3D form into three 1D operations is referred to as the method's *separability*.

For fields with unit cubes (of size 1 in all dimensions):

$$\begin{aligned}
\mathbf{V}(x,y,z)_{linear} \quad = \quad & I(\lfloor x\rfloor,\lfloor y\rfloor,\lfloor z\rfloor)\cdot(\lceil x\rceil-x,\lceil y\rceil-y,\lceil z\rceil-z)+\\
& I(\lceil x\rceil,\lfloor y\rfloor,\lfloor z\rfloor)\cdot(x-\lfloor x\rfloor,\lceil y\rceil-y,\lceil z\rceil-z)+\\
& I(\lfloor x\rfloor,\lceil y\rceil,\lfloor z\rfloor)\cdot(\lceil x\rceil-x,y-\lfloor y\rfloor,\lceil z\rceil-z)+\\
& I(\lceil x\rceil,\lceil y\rceil,\lfloor z\rfloor)\cdot(x-\lfloor x\rfloor,y-\lfloor y\rfloor,\lceil z\rceil-z)+\\
& I(\lfloor x\rfloor,\lfloor y\rfloor,\lceil z\rceil)\cdot(\lceil x\rceil-x,\lceil y\rceil-y,z-\lfloor z\rfloor)+\\
& I(\lceil x\rceil,\lfloor y\rfloor,\lceil z\rceil)\cdot(x-\lfloor x\rfloor,\lceil y\rceil-y,z-\lfloor z\rfloor)+\\
& I(\lfloor x\rfloor,\lceil y\rceil,\lceil z\rceil)\cdot(\lceil x\rceil-x,y-\lfloor y\rfloor,z-\lfloor z\rfloor)+\\
& I(\lceil x\rceil,\lceil y\rceil,\lceil z\rceil)\cdot(x-\lfloor x\rfloor,y-\lfloor y\rfloor,z-\lfloor z\rfloor)
\end{aligned} \tag{2.3}$$

where $I(x,y,z)$ gives the voxel value at location x, y, z in Euclidian space.

Trilinear interpolation therefore produces a value that is linearly interpolated from the surrounding voxels, and is often considered as the best tradeoff between computation and reconstruction quality. Accelerated trilinear interpolation has been offered for some time in graphics hardware supporting 3D texture capabilities.

Second-Order : Tricubic Interpolation

Tricubic Interpolation makes some attempt to ascertain the shape of the data by fitting a cubic function to its surrounding samples. The one-dimensional case requires four weighted neighbouring samples – therefore, the three-dimensional case requires 64. For a given offset t, the cubic filter:

$$k_{B,C}(t) = \frac{1}{6}\begin{cases} (12-9B-6C) \mid t \mid^3 + (-18+12B+6C) \mid & \text{if } \mid t \mid < 1 \\ x \mid^2 + (6-2B) & \\ (-B-6C) \mid t \mid^3 + (6B+30C) \mid t \mid^2 & \text{if } 1 \leq \mid t \mid < 2 \\ +(-12B-48C) \mid t \mid + (8B+24C) & \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

gives the weighting, of t, as shown by Mitchell and Netravali [MN88]. Different values of B and C give different splines, e.g. $B = 0$, $C = 0.5$ is a Catmull-Rom spline. The interpolated value is now the sum of all weights:

$$\mathbf{V}(x,y,z)_{cubic} = \sum_{i=-1}^{2}\sum_{j=-1}^{2}\sum_{k=-1}^{2}\left\{\begin{array}{ll} I(\lfloor x \rfloor, \lfloor y \rfloor, \lfloor z \rfloor) & \cdot k((x-\lfloor x \rfloor)+i) \\ & \cdot k((y-\lfloor y \rfloor)+j) \\ & \cdot k((z-\lfloor z \rfloor)+k) \end{array}\right. \tag{2.5}$$

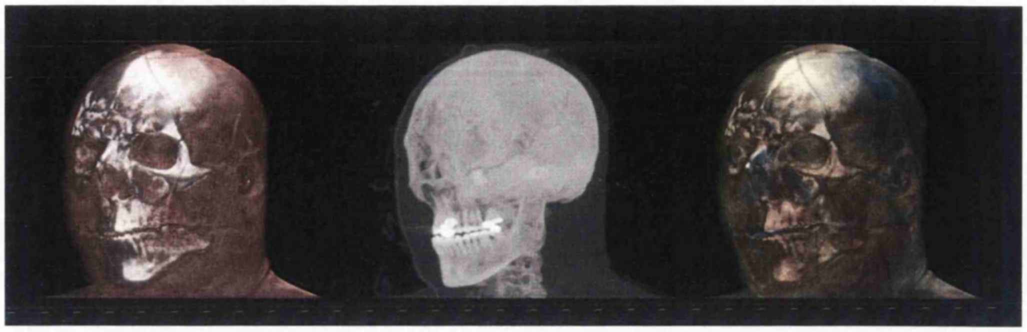
Trucubic interpolation gives superior quality due to its higher-order nature, but at the expense of a large number of data access operations and computations on that data; the computation however can be reduced up by the use of precomputed lookup tables. In general, cubic interpolation should only be used therefore where visual quality is of the utmost importance. Cubic interpolation in general is not yet offered natively on graphics hardware at the time of writing, but it is feasible to implement such a reconstruction filter in real-time as in [SH05] where the authors implement bicubic interpolation using 1D texture lookups.

2.3.3 Transfer Functions and Compositing

Due to the nature of the acquisition of volume data, the samples are often single scalar values representing density or some other kind of measurement. It is desirable for direct volume rendering to establish a mapping between these scalar values and colours, and this mapping is referred to as *classification* [THB$^+$90]. For example, high values in a CT dataset are likely to be bone, with flesh and muscle in the lower ranges. A good visual cue would be to colour the bone an off-yellow colour, and the skin a fleshy colour.

Because the volume dataset is three-dimensional, and the framebuffer two-dimensional, it is apparent that more than one voxel will map to the same framebuffer pixel for sufficiently large datasets. The process of blending the samples together into one final colour is called *compositing*.

One of the main strengths of direct volume rendering is the ability to ascertain internal texture information from inside an object – this can be achieved by setting parts of the volume to be semitransparent. Therefore, alongside colour mappings, opacities can be additionally



(a) Lookup Table (b) MIP (c) RGB Volume Texture

Figure 2.5: Compositing Methods

mapped to the scalar values so that these voxels can be blended together to form a final pixel colour. The opacities are specified as normalised values in $[0, 1]$, with values of 0.0 meaning fully transparent, 1.0 meaning fully opaque, and everything inbetween as various degrees of semitransparency on a linear scale.

The classification process can occur at different stages in the volume rendering process – either before the data is rendered (pre-classification), or during (post-classification):

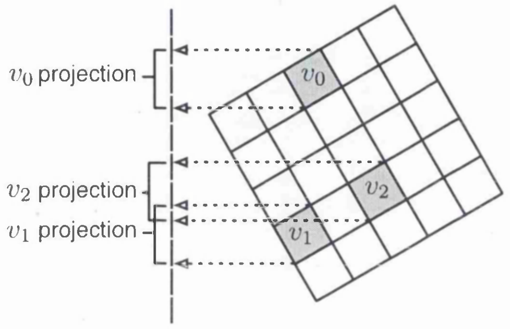
- *Pre-classification* approaches compute the voxel colours and opacities and interpolate these values using a suitable reconstruction filter (such as trilinear interpolation). The resulting interpolated colour and opacity is then used in the compositing operations. These values can be pre-computed before rendering; however, this approach suffers from the potentially large overhead of storing the colour/opacity values instead of a simple scalar that can be mapped to the values at runtime. In addition, changing the classification transfer function at runtime is not possible without rebuilding the entire volume.

- *Post-classification* approaches interpolate the original voxel scalar values from the dataset first, and then compute the colour and opacity based on the interpolated value. As well as being more space-efficient in most cases, post-classification provides a better reconstruction due to the availability of the original signal from the volume (e.g. in a CT dataset, the density values) for input into shading calculations.

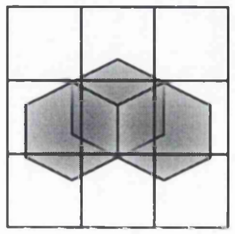
It is not true, however, that such a mapping is always necessary. One compositing method often used for medical datasets is called *maximum intensity projection* (MIP) devised by Wallis *et al.* [WMLK89] and further developed for volume rendering by researchers such as Avila *et al.* [ASK94] who used the method for the visualisation of nerve cells. Figure 2.5(b)) shows a MIP rendering of the Visible Human's head. In this compositing scheme, the highest sampled value falling on a pixel is chosen as the final pixel colour, instead of compositing the results of all samples found along the ray. MIP rendering is commonly used for medical datasets as it allows for a relatively simple means of identifying the high-density structures of the anatomy without having to specify a transfer function.

2.3.4 Projection Methods

Forward-projection methods for direct volume rendering project each voxel onto the view plane based on the current viewing parameters. Each voxel may map to more than one pixel in the final image, and each pixel may have more than one voxel projected onto it. There are a couple of issues that must be addressed with such a technique before an accurate image can be obtained.



(a) Voxels being projected to the screen



(b) Three voxels in image-space

Figure 2.6: A Forward-Projection algorithm acting on voxels v_0, v_1 and v_2

First, as shown in Figure 2.6, a single voxel can map to more than one pixel in the image. Once the transformation matrix is applied to the voxel's position to get the image-space coordinate, the result is a single coordinate. A mechanism must therefore be implemented to spread the effect of the voxel to neighboring pixels, such that the final projection is an accurate representation of the shape of the voxel as viewed from the view plane. In addition, the depth of the voxel at each pixel will differ for non-grid aligned views.

The second issue is that voxels of differing depths in image space are likely to affect a single pixel. This can be seen in Figure 2.6 where voxels v_1 and v_2 overlap in image space. All such voxels are processed either in ascending or descending z-order to be correctly blended.

Splatting

Splatting as a forward-projection volume visualisation technique was devised by Westover [Wes89]. Voxels are projected onto the image plane, and the voxel's *footprint* on the image is computed. The name of the method is derived from the analogy of the voxel splatting against against the screen. The footprint of a voxel is its image-space contribution, and is computed from a reconstruction kernel k; the reconstruction kernel is constant throughout the image for orthographic views.

Figure 2.6(b) demonstrates three voxels' contributions to the framebuffer, affecting four pixels. Each voxel's contribution is greater towards the centre, and less towards its edges, and it is these contribution factors that are modelled by the kernel. The final contribution per pixel is integrated along a projected ray from the pixel, and is given as:

$$contribution(x, y) = \mathbf{V}(i, j, k) \int_{-\infty}^{\infty} h(x - i, y - j, w)dw \qquad (2.6)$$

where h is the reconstruction kernel.

Westover precomputes a table of reconstruction values rather than performing 1D z-integration of the voxels, which increases the efficiency of the algorithm. A guaranteed front-to-back or back-to-front traversal for accumulation of the voxels can be obtained using an octree (octrees are discussed in Section 2.4.1), which means that voxels can be blended in the order they are processed. Splatting can be considered as a special grid-aligned subset of point-based rendering, first devised by Levoy and Whitted [LW85].

Incremental improvements have been made to the Splatting algorithm since its introduction including by Westover himself [Wes90]. Mueller and Yagel provided Splatting with a perspective-correct implementation [MY96] by combining Splatting with a raycasting approach to give a novel hybrid volume renderer. In this approach, rays are fired from each pixel in the image and are intersected with the splats which essentially hang in mid-air. Once an intersection takes place, the data from the splat is gathered from a precomputed table and used in the compositing equation.

Improvements to the problem of aliasing were discussed by numerous authors such as Zwicker *et al.* [ZPvBG01a], who introduced Elliptical Gaussian Average (EWA) splatting to solve the problem of aliasing by introducing a new splat primitive – the elliptical Gaussian filter. The use of this filter is shown to reduce the blurring effect that is apparent with the standard splat filters used by Westover, at the expense of additional computation; though this problem has been partially rectified by subsequent papers detailing GPU-based splatting schemes [CRZP04, BHZK05]. More detail on GPU implementations of Splatting and point-based rendering is given in Chapter 4.

Cell-Projection

Cell-projection techniques decompose the volume dataset into multiple primitive cells that can be efficiently rasterised and composited. Shirley and Tuchman [ST90] discuss the Projected Tetrahedra algorithm, in which the volume is decomposed into tetrahedral cells (and further into triangles for efficient rasterisation) which are then rasterised in depth order using the painter's algorithm and composited into the framebuffer. Wylie *et al.* [WMFC02] implement the original Shirley and Tuchman algorithm on the GPU using vertex shaders to compute the projection of the tetrahedra onto the viewing plane using *basis graphs* to represent the topology.

The majority of cell-projection algorithms for volume rendering give GPU implementations, and for this reason a more complete discussion on Cell-Projection techniques is given in Chapter 4.

2.3.5 Raycasting Methods

Backward-projection techniques for volume rendering discover the final pixel colour for each pixel in the image. This is most commonly achieved using a *raycasting* method. Raycasting methods fire parametric rays from each pixel into the volume data and sample at regular intervals (see Figure 2.7). A ray is defined as a pair P_v, D_v where P_v gives the ray's start position (a pixel in the image plane), and D_v gives the normalised direction of the ray.

The first ray casting algorithm was devised in 1968 by Arthur Appel [App68]. Each pixel has a ray associated with it, which is tested against the objects in the scene for intersection. The frontmost object is then chosen and shaded accorded to its surface properties and lighting conditions. An important extension of this idea was *ray tracing*, devised by Whitted in 1979 [Whi79]. Ray tracing takes the ray paradigm further by calculating further rays bouncing off the surface of objects in a recursive manner, allowing for effects such as reflections and shadows. Ray tracing today remains a popular method for generating high-quality images with complex effects such as global illumination. Unfortunately, the computation time for most raytraced images is very high due to the number of rays fired that further spawn thousands more. This computation however can be partly parallelised and even hardware accelerated using either specialised raytracing hardware or consumer graphics cards [PBMH02].

Each ray can sample the volume dataset many times, and a compositing process determines the final colour for a pixel belonging to the ray.

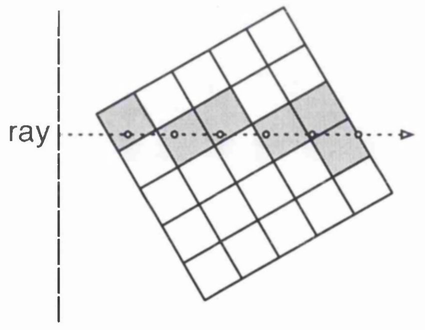


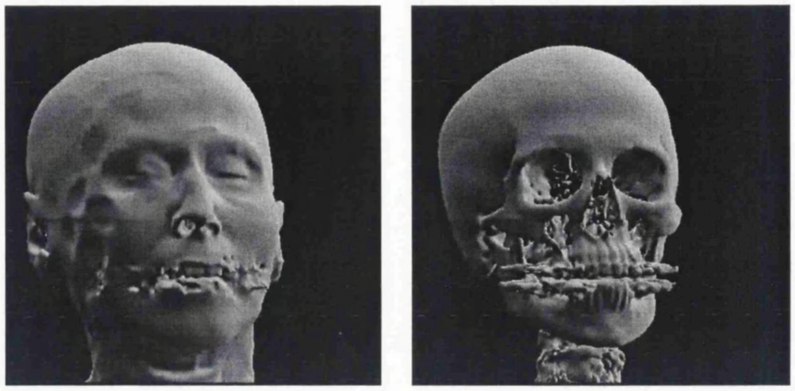
Figure 2.7: A Backward-Projection raycasting algorithm fires rays from the image plane into the volume data, sampling at regular intervals.

Voxels that are more opaque will block more of the light of voxels laying behind in image space, which is how we expect semitransparent objects to behave in the real-world. This gathering of light along the path of ray through the dataset is given by the *volume rendering integral*.

If the samples are gathered in a front-to-back manner, then the process can be formalised by the integral [DCH88]:

$$pixel(R) = \int_{R_{start}}^{R_{end}} C(s) e^{-\int_0^s \alpha(x)dx} ds \tag{2.7}$$

$$pixel(R) = \int_{R_{start}}^{R_{end}} C(s) e^{-\int_0^s \alpha(x)dx} ds$$



(a) Isosurface value : 85 (b) Isosurface value : 120

Figure 2.8: Two isosurfaces rendered directly from the CT head using Jones' Direct Surface Rendering algorithm [JC94b]

where C(s) denotes the emitted light (the *colour*), and α denotes the opacity. The ray direction in this case is ds.

Because computers fundamentally cannot deal with continuous functions, a discrete approximation to the volume rendering integral is obtained from the Riemann sum, with the equation giving the Porter and Duff 'over' operator [PD84]:

$$I(a,b) = \sum_{i=0}^{n-1} C_i \alpha_i \prod_{j=0}^{i-1} (1-\alpha_j) \qquad (2.8)$$

where α_i gives the opacity value at sample point i, and C_i gives the colour. For n discrete samples along the ray, the alpha and colours are composited. The more densely sampled the ray is, the more accurate the resulting colour will be. More samples can be taken by reducing the distance between samples, at the expense of a higher amount of computation.

The above equation evaluates the volume dataset by firing a ray from the viewer into the scene, compositing samples with the previous sample. Back-to-front compositing reverses the direction of the ray, firing towards the viewer. The compositing operation must be modified to take this into account. In general, front-to-back compositing is preferred by most volume rendering implementations as it allows for a group of simple but important optimisations known as *early ray termination* optimisations (see Section 2.4.4).

2.3.6 Direct Surface Rendering

A Direct Surface Rendering method was devised by Jones [JC94b] to allow for efficient isosurface rendering within a volume without converting to an intermediate representation. The method is conceptually similar to that of Levoy's 'isovalue contour surfaces' [Lev88].

A pre-computation step first marks all cells containing the given isosurface – called *transverse* cells. The transversity of a cell can be established by first marking those voxels which are 'inside' the object with 1 (i.e. above the isosurface threshold τ), and those which are not with 0. These values can be stored efficiently as one bit per voxel. A voxel cell is now detected to be transverse if any of the bits of its eight voxel neighbours are not equal; this can be efficiently computed by concatenating the bits and treating the resulting byte as an unsigned char data type – if the value v satisfies $0 < v < 255$ then the cell is transverse.

Rays are fired into the volume dataset as with conventional volume rendering. Non-transverse cells are skipped. For each transverse cell intersected, the actual location of the isosurface within the cell along the ray can be computed using interpolation. First, the value at the point where the ray enters the cube, α is computed, followed by the value where the ray exits, β, using bilinear interpolation of the four neighbouring voxels. The final offset γ of the isosurface along the ray can be computed as:

$$\gamma = \alpha + (\beta - \alpha)\frac{\tau - \mathbf{V}(\alpha)}{\mathbf{V}(\beta) - \mathbf{V}(\alpha)}$$

The computation of shadows with the algorithm is also possible [Jon97]. Figure 2.8 gives two example images of the CT head rendered using the Direct Surface Rendering algorithm. Figure 2.8(a) shows the head rendered at an isosurface value that represents the skin isosurface, and (b) shows the bone isosurface.

2.3.7 Hybrid Approaches

Mroz *et al.* [MKG00] introduced an object-order raycasting system, whereby pre-preprocessed cells of the dataset are selectively raycasted for maximum intensity projection. Mora *et al.* [MJC02] extended this idea further by giving an object-order raycasting algorithm, whereby an octree is used to project nonempty cells to the screen. The projected cells are individually raycasted with full compositing and lighting computations, using rays that are pre-computed for each given cell; these rays are re-computed when there is a change in viewing parameters (for this reason, the algorithm is limited to orthographic projection). A regular sampling for each ray is ensured by taking into account the depth of its cell intersection. The authors give a very efficient implementation that takes advantage of hierarchical occlusion maps to avoid processing hidden octree cells. Mora and Ebert [ME05] give a similar scheme for the production of MIP renderings of volume data.

2.3.8 Lighting

Lighting is an important visual cue in computer graphics, giving the viewer important information on the topology of objects and improving the aesthetic appeal of the image. Lighting in volume rendering was first discussed by Höhne and Bernstien [HB86], based on work by Chen *et al.* [CHRU85] who used the *central differences* method to compute a discrete estimate of the voxel gradient normal as:

$$\begin{aligned} N_x &= \mathbf{V}(x+1,y,z) - \mathbf{V}(x-1,y,z) \\ N_y &= \mathbf{V}(x,y+1,z) - \mathbf{V}(x,y-1,z) \\ N_z &= \mathbf{V}(x,y,z+1) - \mathbf{V}(x,y,z-1) \end{aligned} \tag{2.9}$$

Once computed, the gradient normal is normalised using standard vector normalisation to produce a unit vector.

In volume data obtained by means of medical imaging methods (such as CT/MRI scans), the normals can often be of poor quality and require a preprocessing step to improve them. This is the case particularly when volume datasets are downsampled to a lower resolution using a poor quality filter, or when attempting to compute normals deep within a complex solid object when computing refraction [Rod03]. A filter such as nonlinear anisotropic diffusion can be used to improve the normals from a volume dataset, at the expense of the computation of the filter and / or storage of the filtered normals. Nonlinear anisotropic diffusion is discussed in Section 2.6.

Once the normal of a point inside the volume has been computed, it can be sent to a lighting equation of the implementer's choice. Alternatively, the lighting computations can be pre-integrated into the volume dataset. Beason *et al.* [BGB+06] pre-compute a 3D volume dataset containing illumination data for all isosurfaces within the volume, which includes the computation of shadows falling within a predetermined bounding geometry (such as a Cornell box). This allows for pre-computation of expensive lighting methods such as *global illumination*, which attempts to compute the total light falling on any particular point from all other points in the scene.

2.4 Acceleration Techniques

The field of surface graphics has benefited from years of research into rendering algorithm optimisations. Coupled with the introduction of dedicated graphics hardware, the vast majority of rendering engines written for surface meshes have real-time implementations – even when simply running on the CPU. This is partly due to the large amount of focus on surface graphics as a computer graphics field, and partly due to the inherent simplicity of dealing only with infinitely thin surface representations of objects.

Volume rendering via raycasting is a computationally-expensive operation due to the common requirement of gathering the internal texture information, and the larger datasets involved for input to the rendering algorithm. To alleviate these issues, a number of important acceleration techniques have paved the way for more interactive approaches to volume rendering. This section looks at some of the most important optimisations that can be made for volume rendering algorithms.

Any detail on GPU implementations of volume rendering algorithms is omitted here, and is instead contained in dedicated sections in Chapter 4.

2.4.1 Blocking and Octrees

Blocking techniques attempt to decompose the volume dataset into blocks with attached statistical metadata about the block. An example block metadata would be an indication of whether the block is 'empty' – that is, whether the block contains only voxels that map to 0 opacity using the current transfer function. If a block contains only such data, then it is safe to ignore such blocks rendering as they will not contribute to the final image.

Octrees are a common acceleration data structure which effectively encapsulate the dataset in a hierarchical manner [KWPH06]. An octree is represented as a tree data structure. The root of the tree represents a cell which contains the entire dataset, and its eight children cells subdivide the space within it. This subdivision process continues down to a desired depth. To accelerate a volume rendering application, each octree node may store a Boolean value that denotes whether the cell is nonempty. This octree compilation process is therefore best achieved from the leaves upwards, as the results from the nodes below can be used at each step towards the root.

Levoy [Lev90a] first discusses the use of octrees for accelerating volume rendering in this manner. Each ray is intersected with the topmost cell, and if the cell's precomputed *nonempty* value is *true*, then each of the cell's children in the ray's path are intersected. This process continues down to the lowest level where the nonempty cell is simply traversed as normal and the values along the ray can be included in the compositing operation.

Octrees can also be implemented (nontrivially) on graphics hardware to accelerate GPU-based volume rendering applications [RV06].

2.4.2 Space-leaping

Many acceleration methods for volume rendering employ some kind of space-leaping functionality; a volume rendering algorithm can complete each frame considerably faster if large empty areas of the scene can be simply skipped over. Such areas of the volume can be marked in a preprocessing step to assist the volume renderer decide if it is worth firing rays in that area.

Methods based on occupancy maps [MDHK01] store values in cubes representing whether data exists in that cube; methods based on distance fields [LK04] use the minimal distance to the object of interest. Alternatively, the rendering algorithm can exploit the temporal coherence that exists between successive frames of the rendering process, such as that used by Yagel *et al.* [YS93] where the volume is projected to the image plane before each frame to determine good estimates for new ray starting points based on the previous projection.

2.4.3 Adaptive Termination

For front-to-back ray compositing, *adaptive termination*, introduced by Levoy [Lev90a], can be used to halt the ray's progress if the opacity reaches some threshold near 1.0. The reasoning behind this optimisation is that once the ray becomes very opaque, future compositing operations will not alter the final pixel value substantially enough to warrant its continuation; therefore the ray can be terminated safely without too much impact on the final image.

It is mainly due to this optimisation that the vast majority of raycasting volume renderers accumulate front-to-back rather than back-to-front where such an optimisation would be impossible. Terminating a ray early can also be of benefit to volume rendering algorithms that do not composite semi-transparent samples; such as direct isosurface rendering algorithms, where upon discovering the first isosurface closest to the viewer, the ray can safely be terminated.

2.4.4 Progressive Refinement

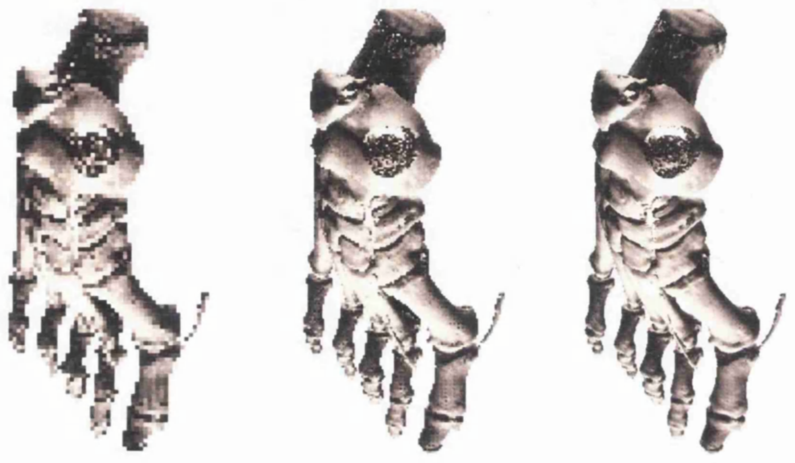

(a) 8x8 block, 0.08 secs (b) 4x4 block, 0.19 secs (c) full, 0.86 secs

Figure 2.9: Progressive Refinement with the CT Foot Dataset

Progressive refinement is a well-established technique in nearly all computer graphics fields, and was first introduced into volume rendering by Levoy [Lev90b]. Progressive refinement schemes first provides the user with a 'rough' render for quick feedback, and then gradually refine this image during 'idle' time by providing more and more detail of the object. 'Idle time' is usually defined to be the period when the user has stopped manipulating the viewing parameters (perhaps when they have released the mouse button that controls the view).

Progressive refinement approaches are particularly well-suited to image-based rendering algorithms such as raycasting, as the render time is heavily dependent on the resolution of the image, and also the refinement scheme can be neatly based upon the number of pixels that are chosen for processing. Figure 2.9 shows three images of the CT foot dataset rendered with a direct volume renderer. Image 2.9(a) was rendered with one ray for each 8x8 block of pixels, 2.9(b) with one ray for each 4x4 block, and 2.9(c) with one ray per pixel. The rendering times for each refinement step are given.

Distance fields, discussed in the next section, can additionally be used to accelerate the rendering process by skipping large areas of the dataset.

2.5 Distance Fields

A *distance field* is a useful representation of the surface of a volume object in which a scalar value at each voxel represents the minimal distance to some surface of interest S. Specifically, a distance field D representing a surface S is defined as $D : \mathbb{R}^3 \rightarrow \mathbb{R}$, and for $p \in \mathbb{R}^3$,

$$D(p) = min\{| \, p - q \, | : q \in S\} \tag{2.10}$$

where q represents a point on the surface of interest S.

Values at each voxel are often stored with higher accuracy than density information datasets, requiring a floating-point value at each voxel for accurate representation. The surface S is a user-defined isosurface value that picks out a particular surface embedded in the dataset; for example, the bone surface in a CT dataset. Distance fields can also be computed for triangular mesh datasets by using a case analysis method to decide whether the point is nearest to each triangle's vertex, edge, or face [Jon95].

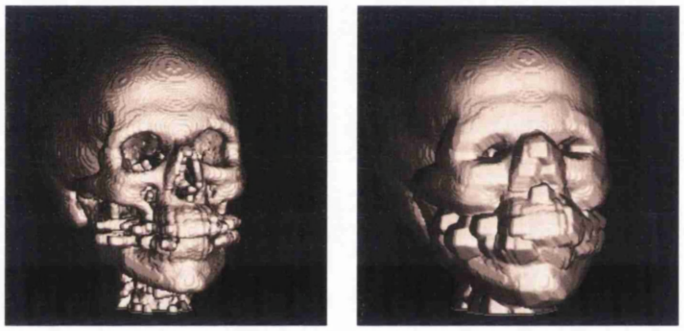


Figure 2.10: Isosurfaces rendered from a distance field. The blocky appearance is intended to demonstrate a low-accuracy distance matrix.

Values in the distance field can be signed depending on whether the point is inside or outside S. This can be implemented by multiplying the right hand side of equation 2.10 by:

$$sgn(p) = \begin{cases} -1 & \text{if } p \in S \\ 1 & \text{otherwise} \end{cases} \tag{2.11}$$

Additionally, the gradient of a point p in a distance field can be computed using central differences, and is orthogonal to the isosurface passing through p.

2.5.1 Applications of Distance Fields

Applications of distance fields include skeletonisation where the distance field is used to find local maxima inside the volume object, as developed by Silver *et al.* [GS99] for volume deformation/animation purposes, but first discussed much earlier by Rosenfeld and Pfaltz in 1966 [RP66] for 2D data. A more detailed analysis of skeleton techniques as used by Silver *et al.* is presented in Section 3.6. Distance fields can be used to apply morphological effects to volume datasets, such as erosion and dilation. These effects can be combined to create opening and closing morphological effects on volumetric data for use with facial reconstruction [Jon01].

Hypertexture, devised by Perlin and Hoffet [PH89], defines a 'soft' region around objects in which a user-defined procedural texture can be rendered. Points in space are defined as being either outside the object entirely, inside, or in the 'soft' region. Satherley and Jones [SJ02] introduced hypertexturing using a distance field to define this soft region. Using a distance field means that even complex objects can be hypertextured, as the task of computing the current region is reduced to simply looking up the value in the distance field. Miller and Jones [MJ05] have demonstrated real-time hypertexturing on the GPU, considerably cutting down the rendering time for visualising hypertexture effects on volume objects. The method loads in a distance field as a 3D texture and performs the hypertexture shading in the fragment shader.

An important observation is that alongside the distance information, any attributes of the surface can additionally be stored. Merely knowing the distance to the surface at any particular point p does not provide any kind of link to the actual minimally-distant point on the surface. Breen and Mauch [BM99] create distance fields of surface-based CSG scenes using a scan-conversion approach, storing additionally the colour value associated with the minimal distance. During the rendering process, the colour values are used to shade the point on the isosurface.

Jones [Jon96] gives an algorithm for producing volume data from a triangular mesh. The algorithm uses a distance field as a smooth function in the region of the surface. The worst case complexity for computing the distance field of size xyz from a set of n triangles is $xyzn$, so the method employs a number of acceleration techniques. Firstly, the author discusses methods for improving the efficiency of the point-to-triangle distance computation by converting the problem to a 2D one [Jon95]. Secondly, an observation is made that a large number of triangles can be eliminated from distance computations by observing that if they are sufficiently far away enough, the true distance does not need to be computed and instead the comparisons can be limited to within a bounding box. Finally, the author gives an optimisation specific to the Direct Surface Rendering algorithm, which has been discussed in Section 2.3.6.

Distance fields can be used to accelerate ray traversal through a volume dataset during rendering. The *Proximity Clouds* method introduced by Cohen [CS94] computes a distance field for the object to be rendered, and uses this distance information to skip large parts of the dataset; if a distance value of d is found at the current ray sample point, then it is safe to move forward along the ray by distance d since the ray is guaranteed not to hit the object before moving that distance.

2.5.2 Distance Transforms

The distance field for a volume object can be computed naively with n^2 complexity. First, all voxels lying on the surface of interest are initialised to 0 and the remainder to infinity. For each voxel in the distance field, the distance to the closest surface voxel is computed. Clearly this approach is too computationally expensive for general use even with octree acceleration [JBŠ06, Sat01], with running times sometimes counted in days rather than hours.

Due to computational complexity of the naive algorithm, research has been focused on fast approximations of the discrete distance field computation. *Distance transforms* were first introduced by Rosenfeld and Pfaltz [RP66], where the authors used the distance information to compute the skeleton of an object of interest in a 2D image. *Distance propagation* methods have an initialisation phase, where a subset of the voxels are initialised to 0, and a propagation phase, where these distance values are propagated to the outer edges of the dataset.

Chamfer distance transforms store a scalar value representing the distance to the nearest point. During propagation, the value for a new voxel is decided based on its neighbours and a distance template. *Vector* distance transforms store a vector pointing to the closest point, and new vectors are decided based on its neighbours and a vector template.

Chamfer Distance Propagation

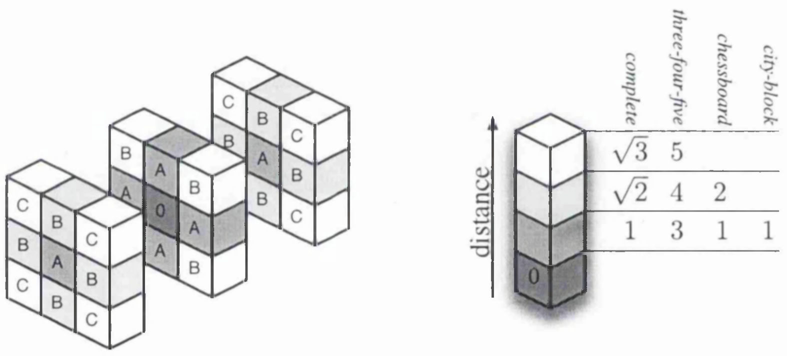


Figure 2.11: Chamfer matrix and a selection of 3x3 matrix values

Chamfer distance propagation techniques propagate local distance values outwards from the surface to the edges of the dataset. Before the main propagation stage, all surface voxels are initialised to 0, and the remainder to ∞:

$$D(p) = \begin{cases} 0 & \text{if } p \in S \\ \infty & \text{otherwise} \end{cases} \tag{2.12}$$

Two passes are now made over the field, one from the top-front-left to bottom-back-right, and the other in reverse. For each point p, and given a distance transform matrix M, the current point is initialised to the minimum distance from local information. The distance transform matrix provides a template of distance values that can be used to add to the current distance value at each point in the algorithm. The choice of transform matrix depends on the accuracy that is required. Figure 2.11 gives an illustration of a distance matrix, and a variety of values with varying accuracy.

Algorithm 1 shows both passes of the Chamfer distance propagation algorithm, where inf returns the inferior result, and M_{fp} and M_{bp} refer to the forward and backward pass subsets of the chosen Chamfer matrix, respectively.

Algorithm 1 Chamfer Propagation

```
{Forward pass}
for $z = 0$ to $d_z$ do
  for $y = 0$ to $d_y$ do
    for $x = 0$ to $d_x$ do
      $I(x,y,z) \leftarrow \inf_{\forall i,j,k \in fp}(I(x+i,y+j,z+k) + M_{fp}[i,j,k])$
    end for
  end for
end for
{Backward pass}
for $z = d_z - 1$ to 0 do
  for $y = d_y - 1$ to 0 do
    for $x = d_x - 1$ to 0 do
      $I(x,y,z) \leftarrow \inf_{\forall i,j,k \in fp}(I(x+i,y+j,z+k) + M_{bp}[i,j,k])$
    end for
  end for
end for
```

Vector Distance Propagation

Vector distance propagation techniques differ from distance propagation techniques in that vectors are propagated instead of just distances. The Chamfer distance transform suffers from a lack of accuracy as the distances from the surface increase. Because of this, vector propagation techniques were devised to propagate vectors to the surface rather than distances; at the expense of additional storage requirements (at least 3 float components per voxel) and computation time [Sat01]. For a given voxel at (x, y, z) in a vector field, the vector value is given as:

$$D(x,y,z) = min(\mid \mathbf{V}(x+i,y+j,z+k) - (i,j,k) \mid) \; \forall i,j,k \in \{-1,0,1\} \tag{2.13}$$

First introduced by Danielsson [Dan80] for 2D images, the vector distance transform is conceptually similar to the Chamfer transform in that it requires two passes of the data, utilising a vector matrix to decide on new vectors at each voxel. The Efficient Vector Distance Transform (EVDT) was devised by Mullikin [Mul92] and performs six passes of the dataset. The EVDT offers a reduced storage requirement by requiring that only the vector components for the current and previous slices are stored at any one time – though this is not as much of an issue with today's personal computers typically shipping with at least 1GB RAM. Jones [Jon04] uses the vector distance transform to achieve lossless compression of distance fields by creating a *predictor* that predicts new distance values from two known values in the field. Satherley and Jones [SJ02] use a vector distance field for hypertexturing volume objects.

2.6 Filtering

The filtering of volume data can improve the accuracy of the normals computed for lighting calculations, particularly if more complex lighting computations are required such as refraction [Rod03]. Filtering volumetric data can also vastly improve the performance and stability of volume segmentation algorithms; segmentation algorithms for volumetric data are discussed in the next chapter. Many segmentation approaches (image-based approaches in particular) are very sensitive to noise in the image, so any noise filtered out would improve the segmentation result. Filtering techniques do not necessarily have to remove noise however, a simple edge-detection convolution filter can provide the desired result if the chosen segmentation technique has trouble locating a boundary.

Nonlinear Anisotropic Diffusion

Nonlinear anisotropic diffusion [PM90] is an iterative filtering process designed to remove noise from medical datasets, but preserve edges. The process is given as the following:

$$\frac{\partial}{\partial t}vol(t) = \nabla \cdot c(t)\nabla vol(t)$$

where vol is the volume dataset at iteration t and c is the diffusion function. Typically this function will return higher values for lower voxel gradients. A parameter k is used to tune the results – larger values of k will have the effect of producing a stronger amount of nonlinear smoothing. The gradient of the voxel can be defined simply as the difference in its neighbours in the discrete case.

Figure 2.12 gives a comparison of two different values of k, with the discrete implementation of function c defined for a given voxel intensity i as:

$$I(x,y,z,t)' = t \cdot [c(x,y,z+1) \cdot (I(x,y,z+1) - I(x,y,z-1))]$$

$$c(i) = (-\frac{i}{k})^2 \tag{2.14}$$

The filtering of volume datasets also has other applications in volume graphics, such as the improvement of voxel gradients for rendering with refraction [Rod03]. Sramek and Kaufman [SK00] used a low-pass filter to smooth resulting volume data generated by their voxelisation method, defining a smooth transition between object and non-object.

Convolution filtering can even be performed on the GPU, as in [HE99], where the authors use a slice-based approach. The recent addition of Superbuffers to OpenGL, and the introduction of DirectX 10 bring the ability to render directly to a slice of 3D texture[1], simplifying future implementations. Viola *et al.* [VKG03] implement a similar slice-based scheme, this time performing nonlinear filtering with edge preservation for the smoothing of volume data.

[1]At the time of writing, Superbuffers is a largely unimplemented standard.

Morphological Operations

Morphological operations are nonlinear filters that have roots in image analysis where they are used for noise removal, edge detection and general enhancement [Wol98]. Morphological operators involve placing a 3D structural element (or *kernel*) defined as 3D grid of values onto each voxel and performing some operation on each surrounding voxel based on the data from the structural element.

For a given object X and structuring element B, The two basic operators used are *erosion*, defined as:

$$X \ominus B = \{x \mid B_x \subset X\}$$

and *dilation*, defined as:

$$X \oplus B = \{x \mid B_x \cap X \neq \emptyset\}$$

In addition, more useful operators can be built from these basic operators, namely *opening* and *closing*. Closing is a dilation followed by an erosion: $X \bullet B = (X \oplus B) \ominus B$, and opening is an erosion followed by a dilation: $X \circ B = (X \ominus B) \oplus B$. The closing operator results in the eventual closure of holes in the data, and the opening operator results in the removal of any small objects.

Jones [Jon01] uses morphological closure on a CT skull dataset by mapping points from a morphologically closed skull to a reference head; giving an excellent demonstration image of the closure operator applied to the CT skull many times. Lürig and Etrl [LE98] use morphological operators for facilitating the specification of transfer functions for volume rendering by performing morphological analysis on the datasets. Hopf and Ertl [HE00] show the possibility of implementing morphological operations on volume data on the GPU by performing three passes over the volume dataset and using the native blending capabilities of the card to compute the filtered values.

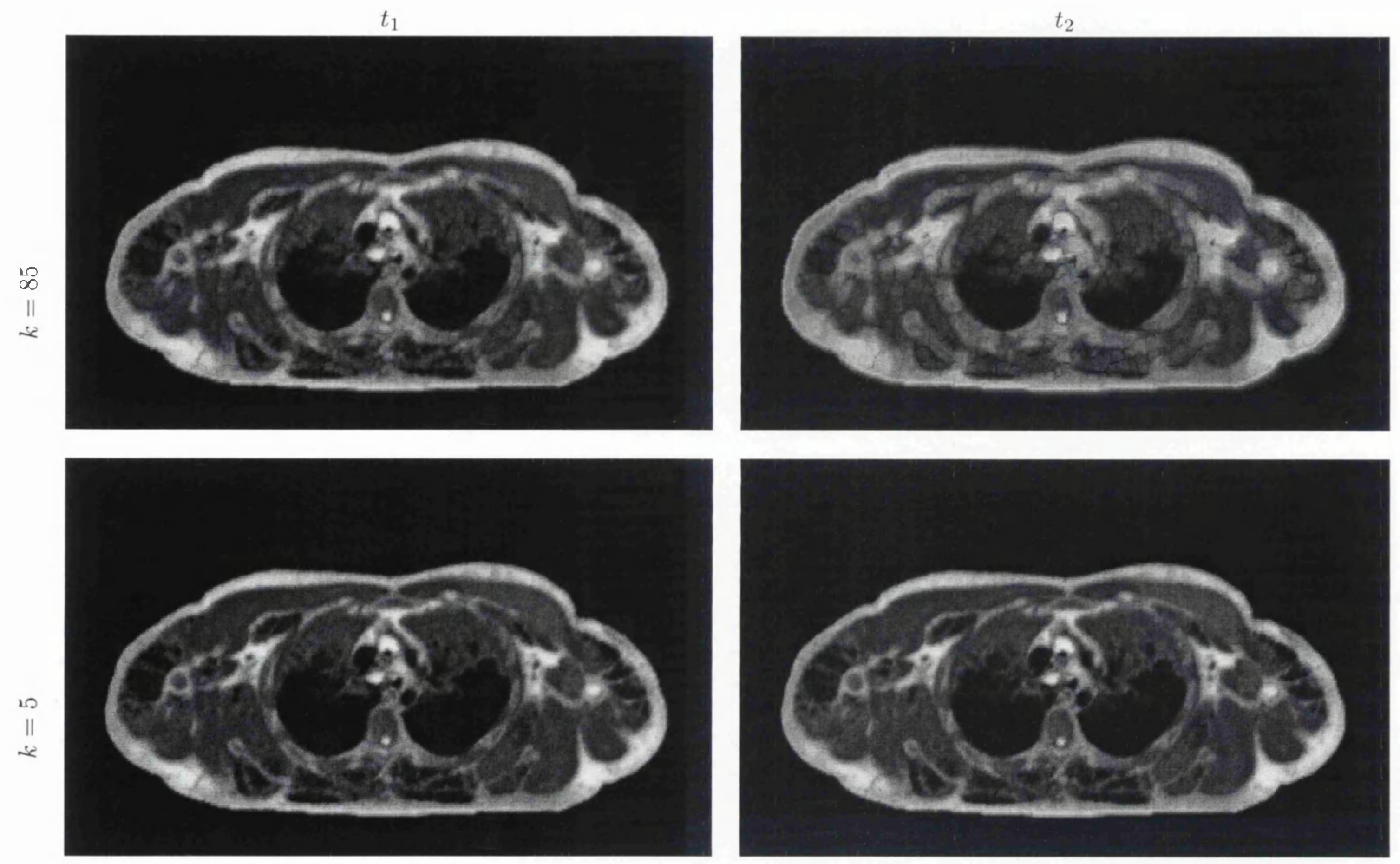


Figure 2.12: A comparison of two different k values for nonlinear anisotropic diffusion filtering on a slice of the Visible Male dataset. Since the filter is iterative, we give snapshots at two time intervals across the x-axis.

2.7 Summary

This background chapter has introduced the field of Volume Graphics and some of the important concepts and algorithms that exist within. It has been shown how Volume graphics differs from the more popular field of surface graphics; the former represents objects as samples in space, whereas the latter represents only the surface of objects. Volume datasets offer many advantages over surface-based models, such as a conceptually simple data structure and the availability of internal information. The main disadvantage with volume datasets it that the necessity of representing so many potentially empty sample points gives often extremely large datasets. In addition, the field of surface graphics is blessed with the most attention, and thus receives dedicated consumer hardware acceleration.

A review of the data acquisition techniques was first made in this chapter, detailing how real-world objects are scanned to produce volume datasets. The details on how these volume datasets are rendered was reviewed, including methods for converting volume data to a surface-based representation for surface renderers.

Chapter 3

Volume Deformation and Animation

Contents

3.1 Introduction

Computer animation has seen a large growth in popularity in the past two decades, mainly due to the entertainment industry's demand for computer-generated films and animated graphics in advertisements. To *animate* means to 'bring to life' – and it is this notion of animation that most people are familiar with. In the field of computer graphics however, animation is not restricted to this connotation; animations produced by computer can be of scientific simulations and thus give additional meaning to aid the viewer in understanding what is being simulated.

Deformation is the act of changing the shape of an object. The deformation of a model over time can be classed as an animation, yet the animation of a scene can consist simply of translations, rotations etc, of the components of the models that the scene comprises; in this case, the objects are not deformed but merely modified in a linear manner. By *linear*, we imply that the transformation of each object can be defined by some single matrix; that is, the translation of each individual point does not depend on where the point is in Euclidean space. This is the definition of deformation given by Barr [Bar84]. Deformation and animation are individual areas of research in themselves, and yet both areas are implicitly linked by similar algorithms and goals.

Although the main focus of this chapter (and indeed, this thesis) is deformation, the fact that animation and deformation are linked in this manner necessitates a discussion of both. This chapter therefore begins by looking at the history of computer animation in general, and discusses various techniques used for controlling such animations. The chapter then continues with a look at the technical issues involved with volume deformation, as well as a review of the most important research conducted in the area. To finish, an in-depth discussion on volumetric segmentation methods is given.

3.2 Computer Animation

Much of the early work on 3D computer animation was focused on bringing the basis of hand-drawn animation, pioneered by Walt Disney, to the computer [Las87]. Traditional techniques, such as *key-framing*, where key stages in a character's animation are drawn and then frames in-between are drawn later, can be implemented algorithmically. Such a method is commonly implemented as *in-betweening* (commonly shortened to just *tweening*), where an interpolation scheme is used to automatically derive the character's pose for the intermediate frames. Varying results can be achieved depending on the interpolation method used; linear interpolation produces often rigid, unrealistic animations, whereas a higher-order interpolation scheme can smooth the motions.

Companies such as Pixar have pioneered many of the modelling and rendering techniques currently used. *Luxo Jr.* was the first film produced by the company, and demonstrated the use of an articulated skeleton, giving a high level of abstraction to the animator to bring a seemingly inanimate object to life. *Toy Story*, widely regarded as the world's first fully computer generated feature-length production, grossed over $354 million worldwide in cinemas according to IMDB [imd].

The control of an animation would be tedious and difficult if the process involved modifying the vertices/voxel that comprise the model over time. Therefore, a variety of control systems exist to assist the animators in defining the movement of the models. The sections following discuss some of the more popular approaches.

3.2.1 Behavioural and Physically-Based Systems

Particle systems were introduced by Reeves [Ree83] to model a wide variety of physically-based phenomena such as smoke and fire. A particle system spawns small objects called *particles* from a fixed or moving spot in the scene. Each particle is given a variety of attributes, such as its colour and opacity, its behaviour, and the amount of time before it is extinguished. The behaviour of a particle is governed by the behaviour function assigned to it, which governs where the particle is located at any one time during the animation. Once a particle has exceeded its life span, it is extinguished and removed. Particle systems were extended by Reynolds [Rey87], who modelled the behaviour of flocks, herds and schools.

Physically-based systems attempt to model the behaviour of an object based on physical laws such as Newton's laws. However, physically-based modelling is most often used for deforming objects by setting material properties and interacting with the model. Related to physically-based systems are *constraint-based* systems, which attempt to (for example) set constraints on the range of movements attainable by a set of muscles for facial animation [Rut99]. Many methods for physically-based animation are carried over to physically-based deformation, and such methods are discussed later in this chapter.

3.2.2 Scripting Systems

The control of animation in early computer animation systems was sometimes achieved using scripting systems such as ASAS (Actor Script Animation System) [Rey82], which provided high-level commands for object manipulation. Although the syntax can be tweaked to be as simple as possible, such an implicit methodology requires animators to be programmers, and offers little in the way of artistic control. It was clear therefore that a more explicit system was required to give this control – but not at the expense of such a low-level methodology as moving individual vertices; this is particularly true with articulated figures such as a human.

In this section, methods for controlling computer animation are discussed. Some methods are high-level in that they are based on abstract concepts (e.g. '*move* the block 5cm north') and others rely on low-level control. It would often seem that an automated, physically-based model would be best suited to animating complex models such as a human being. In the real world however, animators prefer to have more expressive control over the model. This is particularly apparent in cartoon-like animations, where the characters (more often than not) defy the basic laws of physics; for example in the Pixar production *Finding Nemo*, the main characters move through the water almost as if they are unaffected by the water.

The most successful methods therefore, are those which bridge the gap between explicit and implicit control – allowing the animator to be expressive, and yet not give concern to moving individual vertices around. These methods mostly fall under the *control hierarchy* category.

3.2.3 Control Hierarchies and Skeletal Systems

Control hierarchies were devised to bridge the gap between a high level of abstraction, and explicit, low-level control. A control hierarchy allows the animator to move, for example, the hand of a human model and then have the system automatically define the motion for the arm bones based on constraints defined on the joints and muscles.

Kinematics is a branch of mechanics, dealing with the movement of bodies. There are two main methods for achieving skeletal interaction: *forward* and *inverse kinematics*:

- *forward kinematics* involves the animator defining the state of the intermediate joints, with the system computing the final positions for all segments, and
- *inverse kinematics* involves the animator defining just the position of the end points (e.g. the hand), with the system computing the appropriate angles and positions of the

other joints.

It is clear that forward kinematics gives the greatest level of control, but at the expense of the time required to explicitly position the joints. Inverse kinematics removes most of this burden from the animator, at the expense of less control.

A control hierarchy defined for a skeletal system in a human model will possibly have constraints defined, just as joints in the human body do; each joint and muscle in the human body has a limited range of motions. Inverse kinematics therefore allows the system to automatically take care of these constraints. A *skeletal system* is a useful shape abstraction of an object, as it can be considered as a minimal representation of the object to be manipulated.

Chadwick *et al.* [CHP89] developed the first layered skeletal model in which the skeletal model is composed of the *skeletal*, *muscle*, and *skin* layers. The advantage of this separation into layers is that it allows the animator to concentrate only on the skeletal layer during animation, which is relatively simple compared to the other layers. The system automatically defines the deformation of the outer skin layer based on the muscles defined underneath. The muscles in the system are represented by FFD (Free-Form Deformation, discussed later in Section 3.3.2) blocks, which provide volume-preserving squashing and stretching effects, as well as muscle expansion and contraction. An interesting problem is that of how to deal with the joint areas to ensure continuity in the final skin layer, and it is solved by Chadwick *et al.* by defining an additional FFD block at the joint to connect the two FFD blocks connected to that joint.

3.3 Deformation

Deformation is the act of changing the geometrical shape of an object, and is one of the ways in which the temporal behaviour of an object can be changed. Deformation is often a side-effect of computer animation to give greater realism, rather than defining simple linear transformations of parts of the model. Deformation however is not limited to applications in animation; the deformation of graphical objects has many application areas, including:

- *Animation* – as discussed previously, the deformation of soft tissues and muscles in animated characters conveys important visual cues to the viewer and adds either realism or exaggerated, cartoon-like behavior;
- *Surgical simulation* – the planning and training of surgical procedures is a large area, and the physically-based modelling of soft tissues and interaction with surgical tools must be as realistic as possible for accuracy;
- *A Visual Aid* – deformation can add important semantic detail to an object (for example: focus+context schemes [WZMK05]), perhaps by splitting [ISC07, IDSC04] a scanned patient's head in half to reveal the interior structure.

This section explores the use of deformation in the field of computer graphics, without yet considering its application to volume graphics.

3.3.1 Deformation in Computer Animation

The film *Luxo Jr.* by Pixar set not only a precedent for showing exactly how an articulated figure should behave and act; it also set a precedent for artistic expressiveness that had never been seen before in computer graphics. Although the articulated figures comprising Luxo and Luxo Jr. were rigid, the models were given a typical Disney level of expressiveness by deforming them as they interacted with the surroundings; for example, the model is seen to bulge outwards at the base when landing on the ground. This is entirely consistent with the behaviour of a soft body – the actual volume of the base of the lamp was kept consistent as the downforce compressed it. This provides us with a view of the animation as being a *deformation over time.*

The important observation of the above is that although there is a nexus between deformation and animation, there is a fundamental difference between *animating* a model and *deforming* it – animating a model can simply consist of moving its component parts around by translations and rotations, whereas deforming a model means changing the geometrical structure of the model itself. Deforming polygonal meshes is best achieved with a high-level of control; the connectivity between vertices can present a problem if the animator is forced to specify each vertex position manually.

3.3.2 Free-Form Deformation

Barr [Bar84] introduced nonlinear local and global deformation into computer graphics by defining transformations that depend on the position in space on which they act. For example, a twisting effect can be achieved by varying the degree of rotation along the twist axis. Though conceptually simple and mathematically precise, this method requires algorithmic or mathematical specification of the spatial deformation. Specifying deformations in this manner is very much on a trial and error basis, and offers no user interaction. Barr later discusses methods for ray tracing parametric and implicit surfaces [Bar86] by defining the problem as computing the mapping between world and object space.

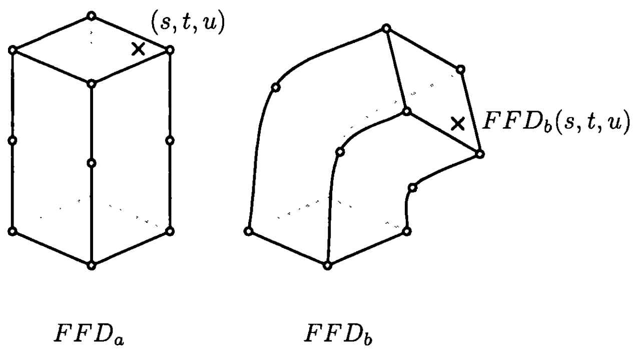


Figure 3.1: Free-Form Deformation

Free-form deformation (FFD) was introduced by Sederberg and Parry [SP86] as a method for specifying spatial deformation for any modelling representation. With an FFD approach, objects are embedded in a space (the *FFD Block*) which is actively deformed, deforming the object inside. The deformations are modelled by manipulating the control points of a parametric model; the author uses trivariate Bernstein polynomials. Once an initial parallelepiped lattice is defined, the lattice-space coordinates are found for all vertices within it. The lattice is then deformed by manipulating the control points of the Bernstein model, and the deformed vertex positions of the object are found by substituting the lattice space coordinates into the Bernstein equation. Volume-preserving FFDs are also possible, and are useful for modelling squashing and stretching on solid objects. Coquillart [Coq90] later extended FFD to deformation in non-parrallelpiped lattices to become EFFD (extended FFD), at the expense of additional complexity in computing the new vertex coordinates for the deformed lattice.

Kurizon [KY95] introduced *ray deflectors* to the raycasting/raytracing pipeline which allow for both global and local deformations of any object representation that can be rendered with a raycasting renderer. A ray deflector is defined as a source of gravity with a given strength and field of influence (its *gravity field*), which has the effect of deforming the path of rays fired through world space. The gravity field is defined as a sphere for simplicity, but the author states that arbitrary shapes are possible. In addition, gravity fields are allowed to overlap. One source of trouble with this approach, although elegant, is that the user must think in terms of the opposite effect; gravity pulls rays towards it, not the intended objects. Placing a gravity source near a box, for example, will produce a noticeable spherical bulge in the box where rays that were destined to miss the box were pulled towards it. Kurizon and Yagel [KY97] later followed up with a GPU implementation of ray deflectors, performing the forward-mapping of the deflection on the CPU and using the GPU to render the resulting piecewise linear approximation.

3.3.3 Physically-Based Modeling and Simulation

Barzel defines physically-based modeling as a modeling scheme in which *"...the behaviour of objects is determined via simulations of physical laws"* [Bar92]. Terrzopoulos *et al.* [TPBF87] first applied physically-based models to computer graphics by simulating flexible materials such as rubber and cloth under the influence of external forces. Physically-based modeling and simulation plays an important role in CG animation productions, ensuring that objects interact with their surroundings correctly, and if necessary, are deformed appropriately. The Pixar production *Finding Nemo* employed extensive Computational Fluid Dynamics simulations to achieve the highly realistic behaviour of the water. The most important role of physically-based modeling is found in scientific simulations.

Physically-based modeling plays a large role in surgical simulation due to the requirement of a high-accuracy simulation of the real world. Surgical simulation is most often used for the training and planning of particular surgical operations, and involves simulated models of organs, tissue, etc., and simulated surgical tools such as the scalpel. Such simulations are commonly undertaken with a haptic input device, so the feedback of forces upon this device must be faithful also. In surgical simulation, soft tissues must behave as they do in the real world, reacting to the influence of external forces (i.e. the virtual tools) and internal forces (stretching, compression, etc); the expense however of such accuracy is usually a high amount of computation involved.

Simulations of real materials often involve the computation of many partial differential equations, which severely limits the real-time implementation options. Reduced accuracy and approximations of the models are often employed, along with various optimisation techniques. The two most common simulation techniques, namely *mass-spring* models and the *finite element method*, are discussed below.

Mass-Spring Models

Mass-spring models are regarded as the conceptually simplest physically-based model [CIJ+07]. A mass-spring model represents the deformed object as a grid of *mass points*, and a set of *springs* which connect the mass points together. The connection of mass points is made not only to neighbouring mass points, but also diagonally-adjacent mass points. At any point in time, the state of the system is defined as the the positions and velocities of the mass points; the forces on each mass point are calculated from the internal forces caused by propagation from neighbouring mass points through the springs. The motion of each vertex based on this information is calculated from Newton's second law of motion, and numerical integration must be applied to solve the system of coupled ordinary differential equations.

Mass-spring models are commonly used in surgical simulation where soft tissue and muscles must behave correctly in order that the doctors obtain a realistic simulation [NT98]. Such systems can be implemented on the GPU as in [GW05] where the authors implement a real-time cloth simulation by solving the system in the vertex shader and writing the new vertices to a vertex array using *render-to-vertex-array* functionality.

Finite Element Method

The Finite Element Method (FEM) is a method of simulating the dynamic behaviour of a deformable object when presented with forces. Because computers cannot deal with the continual nature of such models, FEM discretises the computation and interpolates between samples smoothly to approximate the continuous solution. FEM plays a large role in the simulation of surgical operations as accuracy of simulation is of utmost importance [CIJ+07]. Surgical simulation is useful not only for training purposes, but also for planning a complex surgical procedure in advance of the real thing.

The disadvantage of FEM for such an interactive application as surgical simulation however is its computational complexity; FEM requires a typically very large number of partial differential equations to be computed, making real-time implementations difficult. Many papers have been published attempting to reduce the computation, including simplification of the mathematics [BNC96], offloading some computation to the GPU [BFGS03], developing hybrid FEM/mass-spring models [CDA00] or providing multilevel approaches [DDCB01].

3.4 Deformation of Volumetric Data

A volume dataset offers a distinct advantage over a surface representation of an object: the ability to view the full internal texture information with semi-transparency. Depending on the rendering algorithms used, transparency with surface-based graphics can be a complex procedure; with the traditional OpenGL/DirectX rasterisation pipeline, any semi-transparent surfaces first have to be depth-sorted in order for the card to perform a correct blending operation for all pixels. Depth-sorting objects however is nontrivial, as special consideration is required when objects intersect.

As discussed in Chapter 2, volume graphics offers simple rendering algorithms that can offer high visual accuracy and full internal texture information, and even interactivity when implemented with optimisations or modern GPUs. Coupled with the increased availability of volume datasets from scanning equipment, volume graphics is a fast-growing field of computer graphics. Unfortunately, it is apparent that the vast majority of deformation techniques available are primarily surface-based and do not take the discrete nature of volume data into account. It would be a great benefit to animators to be able to take advantage of the wide variety of volume models available scanned from real-world objects, complete with full internal texture information, and integrate them into the standard graphics pipeline.

This section explores the issues surrounding volume deformation from a technical standpoint, detailing methods for deforming volume data in the discrete (by treating the volume as a collection of samples) and continuous (by evaluating the deformation in world space) cases.

3.4.1 Difficulties of Volume Deformation

There are several areas of difficultly in working with volume data for deformation purposes. The deformation of volumetric datasets is seen as a more expensive problem than traditional surface-based animation due to the sheer amount of data that is required to be modified – surface-based models only need to represent the boundary of objects via mesh approximations, whereas volume datasets often end up representing much empty space and internal texture that may not even be seen.

Most volume datasets are created by scanning real-world objects; the end result being a large group of samples in a grid with no topological or semantic information. The majority of surface-based models used for animation are created with deformation in mind and thus are created with all geometric and semantic information necessary for intuitive deformation, such as joint/bone connectivity, range of motion and abilities of particular muscles structures, and so on.

The main issues with deforming a discretely sampled object are given below.

- *Lack of geometrical information* – the information governing the shape of the objects contained within a volume dataset has to be inferred with post-processing methods. It is often useful before adding semantic information to such a dataset to be given information on the shape of the objects contained within. Such information however can be approximated using various methods – i.e. the construction of a given isosurface using the Marching Cubes algorithm [LC87], or added by hand.

- *Lack of semantic information* – semantic information with computer models in general is usually added as a post-processing task – once an animator creates the skeletal model and skin for a character, the relationship between joints is defined. Volume datasets are particularly hampered in this respect due to the larger amount of data; adding semantic information for each voxel would result in a time consuming process. Semantic information can be added more easily with segmentation techniques, where portions of the volume dataset are identified quickly as being part of one semantic object or another; for example, the white matter and grey matter in a CT scan of a brain. Segmentation techniques are discussed in detail at the end of this chapter in Section 3.8.

Because of the sheer bulk of data in a volume dataset, deformation methods for volume data will often convert to an intermediate representation of a pre-computed isosurface (using Marching Cubes or similar methods). The deformation of a polygonal mesh (ignoring the deformation methodology used) involves the translation of its vertices. Once these vertices are translated, the new polygons can be rasterised as before using a forward-projection or backward-projection method.

3.4.2 Spatial Transfer Functions

Spatial Transfer Functions (STFs), introduced by Chen *et al.* [CSW+03], define a unified method of specifying the geometrical transformation of every point in a volume dataset. A function $\Phi : \mathbb{E}^3 \rightarrow \mathbb{E}^3$ defines, for each $p \in \mathbb{E}^3$, the spatial transfer of point p. An associated inverse function $\Phi^{-1} : \mathbb{E}^3 \rightarrow \mathbb{E}^3$ defines, for each $p \in \mathbb{E}^3$, the inverse of this deformation such that $\Phi^{-1}(\Phi(p)) = p$. Depending on the complexity of Φ, the inverse can be difficult to obtain automatically. A new volume dataset $\mathbf{V}'$ from a spatial transfer function is therefore defined as, for each $p \in \mathbb{E}^3$:

$$\mathbf{V}'(p) = \mathbf{V}(\Phi(p)) \tag{3.1}$$

Spatial transfer functions can be considered as scalar fields in their own right, and are referred to in this context as *spatial transfer objects*. Spatial transfer objects can therefore be represented as a discrete volume dataset where each point p in the dataset contains a new point p' denoting the spatial transfer of p to p', or in the case of a backward-mapping, p' to p. Related to STFs are SDFs – Spatial Displacement Functions [Isl07], which define the relative displacement of the point instead of a new absolute position.

Figure 3.2 shows two frames from an animation of the CT Carp dataset. The animation was specified with a backward-mapped spatial transfer function that performed a simple inverse bend around the x axis. The texture of the carp was produced by obtaining an image of a carp, and producing a small tessellated image of its skin using the image editing software Paint Shop Pro's tessellation functionality; the texture image was applied to the carp by simply sweeping the texture through one axis. The lighting of the carp was computed using backward-mapped points for central differences, as shown in equation 3.2. The cost for

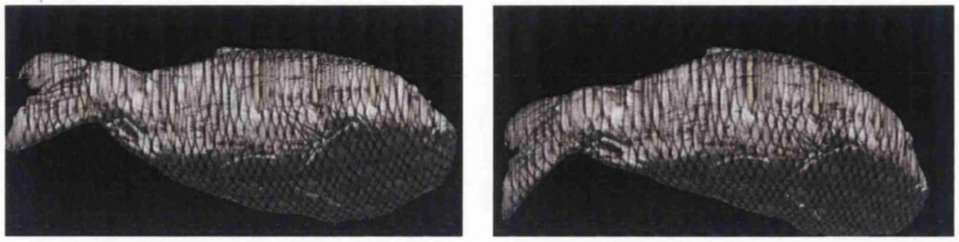


Figure 3.2: Two frames from a backward-mapped Carp animation, specified using a spatial transfer function. The carp is textured with a 2D image swept through one axis.

rendering one frame of the animation on a Pentium 4 3.4GHz using a software raycasting engine was approximately 40 seconds.

The carp animation shows that for simple deformations, spatial transfer functions work well; the function itself was specified in around 5 minutes including tweaking time. However, it does highlight one of the drawbacks of backward-mapped spatial transfer functions in general – the prohibitive cost of rendering. It is not possible to tell in advance, given a point p on the ray, whether p will contribute in any significant way to the final image; i.e. after applying $\Phi^{-1}(p) = p'$, whether p' will land inside the area representing the body of the carp. p' may land outside the carp in a region that maps to zero opacity, or even worse, could land outside of the dataset entirely; it is only after applying Φ^{-1} that this can be known for sure. Various optimisations can be made to the STF to avoid complete evaluation in particular world-space locations, but the problem is still one with no general solution.

3.4.3 Forward and Backward Mapping

The choice of rendering method used for the deformed volume data is directly affected by the underlying choice of mapping direction, and vice versa. In the case of a forward-projection algorithm (such as Splatting, as described in Section 2.3.4), the voxels are projected to the framebuffer in their newly transformed positions. For backward-projection algorithms, the renderer must discover the actual point to sample in the dataset for each sample point on the ray. Assume the existence of a spatial transfer function Φ which takes a point $p \in \mathbb{E}$. The inverse of this function, Φ^{-1} essentially asks the question *"For the current point in the world, what point should be here based on the deformation?"*. This deformation inverse function Φ^{-1} is used with raycasting algorithms for volume rendering.

Figure 3.3(a) shows a forward-projection algorithm operating on a voxel in the dataset with a spatial transfer function. The spatial transfer function ϕ is applied to the dataset voxel to obtain a new voxel position (left-hand side of diagram), which is now forward-projected to the framebuffer. Figure 3.3(b) shows a backward-projection algorithm applying the backward-mapping function Φ^{-1} to a selected sample point on the ray. The resulting point (right-hand side of diagram) is the new sample point to be used in subsequent stages of the rendering process. In the case of direct volume rendering, this point would be sampled and used in the ray accumulation equation.

In order to obtain accurate lighting information, the gradient computed using central differ-

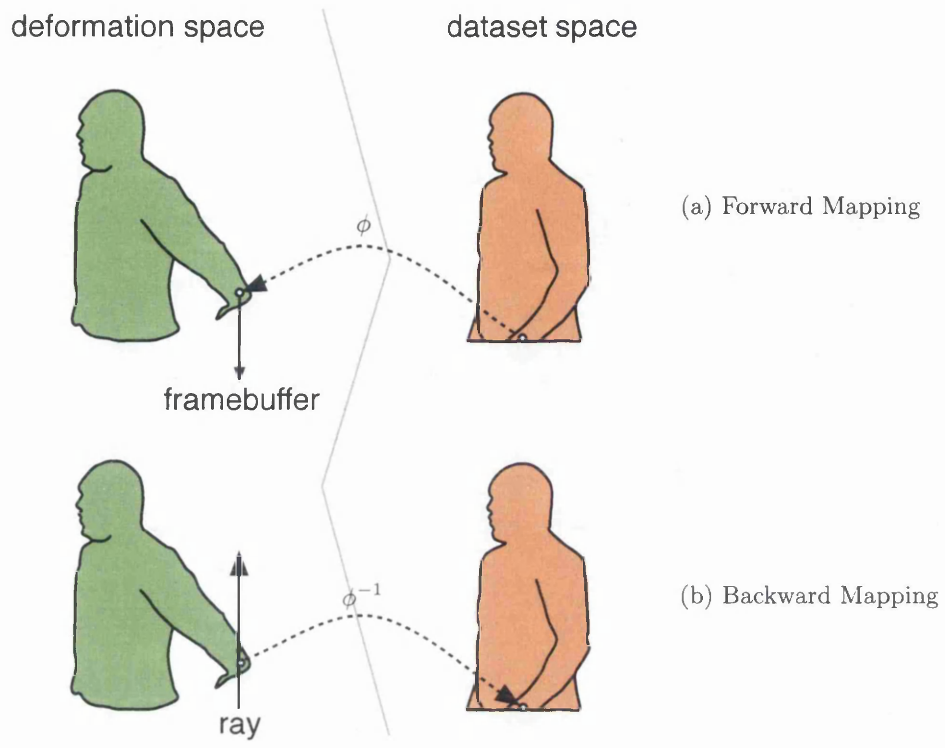


Figure 3.3: Forward and backward spatial mapping

ences must be evaluated in deformed space by applying the Φ^{-1} to each point used in the equation. Equation 2.9 for computing a normal with the central differences technique now becomes:

$$\begin{array}{rcl} N_x & = & \mathbf{V}(\Phi^{-1}(x+1,y,z)) - \mathbf{V}(\Phi^{-1}(x-1,y,z)) \\ N_y & = & \mathbf{V}(\Phi^{-1}(x,y+1,z)) - \mathbf{V}(\Phi^{-1}(x,y-1,z)) \\ N_z & = & \mathbf{V}(\Phi^{-1}(x,y,z+1)) - \mathbf{V}(\Phi^{-1}(x,y,z-1)) \end{array} \tag{3.2}$$

It is apparent from Equation 3.2 that if the cost of the deformation function is high, then the cost of computing an accurate voxel gradient will be especially high, as the deformation inverse function must be computed an additional six times. If the Jacobian of the deformation can be discovered, then a much more efficient mapping of normals can be realised by computing normals in object space and mapping them directly to world space.

There exist a variety of surrounding issues with both the forward-projection (and subsequently forward-mapping) and backward-projection (backward-mapping) of deformed volume data:

- *Forward-projection* – In the case of *forward-projection* rendering, there exists the problem of gaps appearing where voxels are pulled apart, creating holes in the final image. This is due to samples adjacent in world space mapping to non-adjacent pixels in image space. One solution is to create new samples on-the-fly, but determining where to create these new samples could prove difficult and memory-intensive. An image-based solution could attempt to close the gaps in the final image, but the difficulty this time is choosing exactly what data to place in the gaps. A related problem is that of two or more voxels projecting to the same pixel in screen space.

- *Backward-projection* – Raycasting renderers are presented with the difficulty of recovering samples when the volume grid loses its connectivity information. In a structured grid, the position of each voxel is inferred implicitly – for each sample point on the ray, the position in the volume dataset can be calculated simply. The lack of structure caused by any non-linear deformation necessitates additional data to be encoded with each voxel detailing the explicit position of the voxel. However, even with this additional information, calculating the sample point position inside the deformed dataset is nontrivial.

In a structured dataset, the voxels can be accessed in memory implicitly. For example, in a structured, regular 3D grid, a voxel at position $p_{(x,y,z)}$ can be accessed at memory location $p_{(x,y,z)} = data + (z * dx * dy) + (dx * y) + x$. For a dataset placed at the world origin with 1:1 scale, world positions within the dataset boundaries correspond to dataset positions. Affine transformations on the dataset are also possible by transforming world points to object space with transformation matrices. When the dataset has been deformed and therefore de-structured, simple methods for obtaining surrounding voxels for interpolation no longer exist. If such voxels are located, the method of recovering a continuous value based on their positions and values presents a further problem.

3.4.4 Reconstructing Volume Data

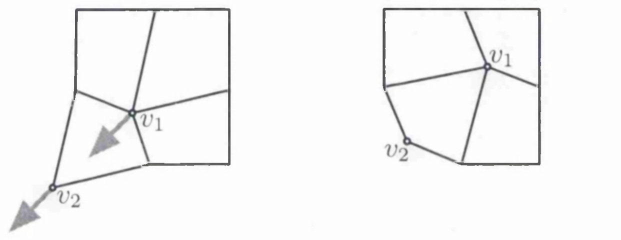
The most important consequence of the issues discussed in the previous section is that there is occasionally a close coupling between the deformation methodology / algorithm and rendering algorithm, unless some form of reconstruction takes place; that is, creating a new volume dataset to be rendered with a standard direct (or even indirect) volume rendering algorithm. A close coupling of deformation methodology can exist where complex data structures are devised to maintain deformation information, which must then be evaluated for rendering. Evaluating the deformed object in world space using a backward-mapping function however frees such restrictions, provided such a function exists and is easy to compute.

The advantage of reconstructing a deformed volume dataset and rendering this new dataset is the complete decoupling between the deformation process and the rendering algorithm, which allows the implementers of both algorithms to create fully generalised solutions without the risk of restrictions caused by the coupling of algorithms. The main disadvantage however is the potentially large storage cost when one considers the animation of large datasets such as the Visible Human dataset, which is around 1GB in its static state. If the Visible Human is moved into a new pose, such as bringing the arms out in front of him, then there is potentially a large amount of wasted space created by the extension of the dimensions of the dataset. A block-based volume representation can partially alleviate this problem, but such schemes then place an additional burden on the volume rendering algorithm to evaluate this data structure.

3.4.5 Deformation Encoding & Dependent Textures

For deformation methods choosing to work directly with the discretely sampled data, an encoding of the world space deformation into a new volume dataset is possible. Using such a system, a renderer can sample each position within the encoding dataset, and use this information to discover the new sample point in object space.

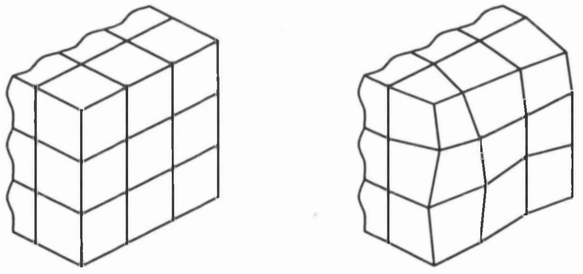
Backward-mapping algorithms can be implemented on the GPU in this manner by using *dependent textures*. A dependent texture is loosely defined as an *indirection texture*, used exclusively to gain new texture coordinates for some other texture (in this case, the volume dataset). The encoded values can represent new absolute texture coordinate positions, additive coordinate offsets, or a set of parameters to be sent to some mapping function.



(a) v_1,v_2 translated (b) Resulting texture coords

Figure 3.4: Deformed Texture Coordinates (2D example)

A system utilising dependent textures was developed by Rezk-Salama *et al.* [RSSSG01] deforming volumetric objects by adaptively subdividing the volume using piecewise linear patches. The user specifies the deformation by picking voxels to drag around. The inverse of this deformation can be approximated by negating the translated voxel offsets for the texture lookup. Figure 3.4(a) shows two voxels v_1 and v_2 being pulled downwards and to the left.



(a) Volume grid (b) Deformed texture coords

Figure 3.5: Deformed Texture Coordinates (3D example)

3.5 Illustrative Deformation and Visualisation

Illustrative visualisation of volume data is a useful tool for providing abstractions of medical data to increase the clarity of the final image and increase the emphasis of particular regions of the data. This section investigates illustrative visualisation techniques with a particular emphasis on techniques utilising spatial deformation to achieve their results.

3.5.1 Focus+Context

The aim of illustrative techniques is to bring objects of interest into main focus, but preserve the overall context; the compliment of this object should also be visible, but brought out of main focus. This technique in general is known as *focus+context* visualisation, and is the subject of much research not just in the graphics community, but in the human-computer interaction community also. Usability First defines focus+context as [Des03]:

> *A principle of information visualization – display the most important data at the focal point at full size and detail, and display the area around the focal point (the context) to help make sense of how the important information relates to the entire data structure. Regions far from the focal point may be displayed smaller (as in fisheye views) or selectively omitted.*

Focus+context schemes need not always resort to spatial deformation. Viola *et al.* [VKG04] implement a focus+context volume rendering algorithm in hardware using a technique called *importance-driven volume rendering* in which regions of the volume are labelled according to their importance and then the most important regions are given priority when visualised by rendering these regions in front of the other regions. Ikits and Hansen [IH] give a focus+context interface for GPU volume rendering with the aim of improving user interaction with volume data. The authors implement an interactive lens for magnifying portions of the volume that the user is interested in, performing the perturbation of texture coordinates in the fragment shader.

3.5.2 Utilising Deformation for Visualisation

Wang *et al.* [WZMK05] implement a focus+context volume rendering algorithm using a variety of different 'magic lenses', similar to Kurzion's ray deflectors ([KY95]). The lenses are semantically-aware in that they can deform the path of rays fired into the object depending on whether the rays are near an important region in the volume. Important regions have the effect of pulling rays towards them, which on the final image has the effect of making these regions appear much larger than they actually are.

McGuffin *et al.* [MTB03] make the case for using deformation methods for improving the ability to view the internal structures of volumetric data, showing that spatial deformation can better assist the user in understanding the object interior over simply using semitransparent transfer functions with direct volume rendering. Their system implements a variety of tools with real-world analogues, such as cutting and peeling tools. The tools are designed to remove large parts of the dataset 'out of the way' to enable the interior of the object to be visible primarily. The tools are also semantically-aware in that the tool's effect differs depending on the value of the voxel, allowing for effects such as splitting the skin away from bone. The rendering algorithm is a simple point-based renderer using forward-projection of the new voxel positions.

A tool developed by Chen *et al.* [CCS06] allows for illustrative visualisations of volume data through tool-based deformations. The tools are designed to cut, peel and dissect the data to allow for illustrations like those typically found in medical texts. The authors give examples of the Visible Human's wrist being cut open to reveal the internal muscle and bone, and the skin of the head being separated from the skull. An important distinction is made between *axis*, *surface*, and *segment*-aligned operations: axis-aligned operations treat space as divided by an axis-aligned plane; surface-aligned operations treat space as divided by some arbitrary surface (e.g. a distance isosurface from a computed distance field); and segment-aligned operations treat space as divided by segmentation operations. The authors additionally give a normal-blending scheme for correcting normals at discontinuities. A GPU implementation is discussed by the authors whereby a backward-mapping scheme operates on the fragments generated by slices aligned with the viewer. The requirement of storing both the volume dataset and the feature mask as a 3D texture imposes a high memory requirement on the technique.

Displacement mapping was introduced by Cook [Coo84] as a simple means of adding surface detail to objects by perturbing points on the surface in roughly the direction of its surface normal. Correa *et al.* [CSC03] give a system for performing discontinuous displacement mapping on volumetric models in which the displacement mapping function is discretised into a 3D dataset and stored on the GPU alongside the volume dataset. A slice-based approach is used to render the displaced volume, allowing the graphics hardware to perform compositing; the slices chosen are based on the bounding box of the displacement volume in world space. A fragment program running on the GPU computes the new sample point p' in the volume dataset as $p' = p + D_k(p)$ where p is the current world-space point and D_k is the displacement texture. Displacements are handled by defining an alpha mask stored as an additional 3D texture that defines the region of discontinuity. As with [CCS06], a normal-blending scheme is used to correct the incorrect normals caused by discontinuities.

Bruckner and Gröller [BG06] devised a similar system for illustrative visualisations, in that the overall context is maintained by keeping non-focus objects in scene. Their methodology is based around 'exploding' segmented portions of the dataset out of view of interesting portions, termed *focus objects* – e.g. a head's skull and outer skin layers can be exploded outwards, revealing the focus – the brain matter. The volume is decomposed into parts using tools such as an axis splitter, and each part is rasterised on the GPU in depth order. The raycasting algorithm correctly skips over areas of the dataset from which data has been moved, and performs a static transformation to backward-map the exploded parts.

Rautek *et al.* [RVG06] give a system for caricaturistic visualisation of volumetric data by specifying points in the volume data corresponding with points in a reference dataset and exaggerating their positions. Their system also provides a GPU raycasting implementation using adaptive refinement to give interactive feedback, with a high quality image generated in around a second.

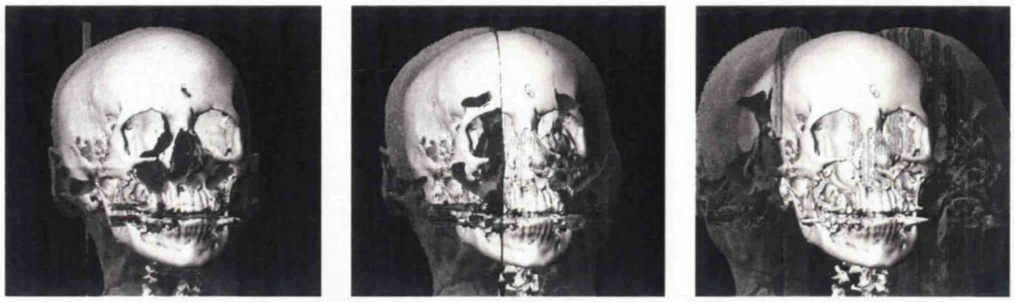


Figure 3.6: A series of frames from an animation of a logical/semantic split of the CT head

3.5.3 Volume Splitting

Splitting a volume dataset into two or more components is an effective means of providing an insight into the internal structure of the object, and is discussed in much detail in [Isl07]. Volume splitting has a large application in medical illustration, where, for example, a CT head can be split in half to reveal the complex bone and muscle structures inside; such images are indispensable for learning materials. Volume splitting also has an important application in surgical planning and training, where real-time splitting and rendering methods are required with physically-based deformation; in this case, splitting is nontrivial since splitting the object results in a major change of geometric structure and often results in the object becoming two objects.

Islam *et al.* [IDSC04] present some algorithmic approaches to volumetric splitting. Volumetric splitting is defined to be the transformation of two or more component objects over time such that when the time is zero, the union of all such objects is equal to the original volume dataset. Explicit and implicit splits are the focus of discussion by the authors; explicit splitting involves partitioning the volume dataset into explicit component objects and then applies the transformations to each of these objects, whereas implicit splitting defines the regions in terms of the transformations themselves. For explicit splits, a distinction is made between geometric splits where the volume object is split simply in terms of its bounding geometry (e.g. down one axis), and logical splitting, where semantic information applied to the volume dataset is used to split it (e.g. splitting the skin away from the bone in a CT dataset). The techniques discussed are demonstrated in the form of animations produced using both explicit splitting, where the visible human is shown to be split into geometrical blocks and a particle-like behaviour is attached to each block, and implicit splitting, where the skin and bone are separated in a CT head dataset.

To facilitate easier specification of algorithmic splits, Islam *et al.* [ISC07, Isl07] devised the *STOM* (Spatial Transfer Object Modeler) tool, which allows for both semantic splitting (by loading a *mask volume* which explicitly labels each voxel), and geometrical splitting. The user interface presents the volume dataset as a set of cubes or spheres to take advantage of rasterisation hardware if available. Operations such as cutting, peeling and twisting are specified in real-time by viewing the mesh representation of the object. A final ray casted image is achieved by exporting the operations specified as a *spatial transfer object* (discussed in Section 3.4.2, and then importing this object into an API such as vLib.

3.6 Skeletal Systems

The production of realistic-looking animated figures is one of the biggest areas of research in computer animation today; it is considered by some to be the holy grail of animation to be able to animate humans and animals in a high-level manner. A comparatively small amount of research however has been focused on the use of character-based deformations and animations in volume graphics. This is particularly remarkable when one considers the availability of the Visible Human dataset, which provides an accurate and complete 3D RGB volume model of a human being.

Curve-skeletons [CSM05] can be created automatically from volume data and have many applications, and are considered as minimal representations of an object in that it is possible to reconstruct the object from the skeleton. An analogy for the creation of a curve-skeleton is that of a grass fire: if a fire was started on the boundary of the object and began to move inwards, then the skeletal lines would coincide with the points inside the object where the fire fronts meet and extinguish [Blu67]. Such skeletal segments are useful for many applications, particularly in volume graphics, as will be discussed in coming sections. Hu *et al.* [HHCL01] use curve-skeletons for virtual endoscopy, where a reliable path through the volumetric organ (for example, the colon) can be computed automatically; this path can be used for intuitive navigation through the dataset by placing the camera on the path.

3.6.1 Block-Based Deformation

Prakash and Wu [WP00] implement a volume deformation system called *Young_Man* for the purpose of animating the Visible Human dataset. Their system first employs a segmentation method called *clustering* (discussed later in Section 3.8) to identify the different parts of the Visible Male's anatomy, and separating them from the rest of the dataset. The authors employ this segmentation technique on a dataset-aligned slice-by-slice basis to create disjoint regions for the arms, legs, and torso of the human. Once these clusters are identified, a series of bounding polyhedra are placed around them.

The authors produce an animation from this data by deforming the polyhedra and interior voxel positions using the Finite Element Method (FEM) [CE98], and then revoxelising the dataset in the deformed state using a technique called *voxture mapping* (volume texture mapping). The voxture mapping process involves computing the position of each voxel inside a deformed polyhedron block and spreading its contribution to the voxels in the reconstructed dataset, and the authors give an example parallel strategy for achieving this.

The use of FEM by the system provides a certain degree of realism by ensuring that the bounding geometry is correctly deformed when forces are applied – e.g. that the arm of the model flexes. The process of building the bounding polyhedra from segmentation information and using FEM and voxture mapping for reconstruction is detailed in [PC98].

3.6.2 Volume Animation Applications

Work by Silver, Gagvani and Kenchammana has established a group of techniques for character-based deformation and animation of volumetric datasets. The methods published by the group establish an effective IK-skeleton based method for volume animation that provides not only a method for the automatic computation of the skeleton given a set of parameters, but also a method to reconstruct a volume from the modified skeleton.

Initial work on producing skeletons from volumetric datasets was completed in a masters thesis by Gagvani [Gag97]. This work laid the foundations of the techniques published later by Gagvani *et al.*, such as a follow-up paper establishing a technique to automatically produce a skeleton tree from a volume dataset using a volume thinning technique [GKHS98], with a further paper discussing the thinning technique in more detail [GS99]. The technique, called *Parameter-controlled volume thinning*, uses a distance field to identify voxels belonging to the skeleton of the object. Once a distance field has been created for a surface of interest, the skeleton voxels are identified to be the maxima inside the object. The authors test for the maxima by comparing the voxel's distance value with the average of its 26 neighbours. If $Avg26_p$ gives the average of the 26 neighbours, $Dist_p$ gives the distance value, and t is a thinness parameter, then:

$$Avg26(p) < Dist(p) - t$$

gives the skeletal voxels. t is a thinness parameter that controls how many voxels are captured by the algorithm; higher values of t ensure that less voxels are captured, and lower values result in more voxels being captured, resulting in a thicker skeleton with possibly many redundant voxels. A thicker skeleton however often gives a better result when the deformed volume is reconstructed. The subset of voxels defined to be part of the skeleton (the *skeletal voxels*) contain no semantic information useful to the animator, so a further process attempts to automatically derive a skeleton tree from these voxels. The skeleton tree extraction procedure creates a weighted undirected graph from the voxels. The weight of a connection is based on the similarity of the distance values, and also the distance between the voxels; edges connecting voxels close to each other and with similar distance values have smaller weights. The minimal spanning tree is now computed from this graph to give the final skeleton tree. Now that the skeleton is created, it can be manipulated by the animator in an external package such as Alias [GS00b], where further semantic information such as joint movement restrictions can be added.

Further papers by Gagvani and Silver detail the reconstruction process further [GS00a, GS00b, GS01]. Once the skeleton tree has been deformed, the skeletal voxels are transformed into place around it. A new volume is reconstructed to be rendered by an volume rendering algorithm desired. The reconstruction algorithm works by defining spheres around each skeletal point with radius equal to the saved distance value from the previous stage, and scan-filling these spheres into the new dataset; the new volume is defined to be the union of these spheres. Overlapping spheres can result in artefacts near bends where new spheres overwrite the previous spheres, so voxels with minimal distance to the centre of the sphere are chosen when conflicts occur. The reconstruction method gives good result images when rendered with a standard direct volume rendering algorithm. However, it is clear that the reconstruction phase is not only computationally expensive, but also has a large storage requirement. Datasets such as the Visible Human are often around a gigabyte in size – producing an animation at 24FPS would result in a huge number of intermediate datasets being produced.

Silver, Gagvani and Singh give a real-time rendering algorithm for the deformed IK-skeletons [SSC03], implemented into a tool called *VolEdit*. In this system, bounding boxes are defined around each skeleton segment, with a mid-plane algorithm used to connect the bounding boxes to avoid cracks appearing at joint areas. The system now computes a series of viewport-aligned slices intersecting with the bounding geometry, which are textured with the original volume object. The slices are composited back-to-front by the hardware, using the blending capabilities offered by the hardware and API. Though the mid-plane geometry solves the problem of cracks appearing at joint areas, the solution is still not ideal, as joints should not be treated in purely linear fashion. This observation is noted by the authors, giving an example of a joint being rotated around 90 degrees. A better result would be obtained by smoothing the transition from one segment to another, rather than defining an automatic mid-plane.

Singh and Silver later applied the VolEdit system to a focus+context scenario [SS04], additionally giving a forward-projection algorithm for the rendering stage based on elliptical Gaussian Splatting. As well as allowing for the rigid transformations implemented in the previous system, the authors implement additional focus+context tools such as selective compression (varying the resolution of target areas), selective juxtaposition (splicing sections of volume objects together), and selective highlighting (varying the transfer function of target areas).

3.7 Sculpting and Soft-Body Deformation

The sculpting metaphor is based on the act of deforming objects using tools in the same manner as a potter's wheel and carving tools. Sculpting can also be used as a method of creating new volume datasets from template datasets; if the user begins with a dataset with all voxels set to maximum density, they can carve the dataset into the desired shape. Galyean and Hughes introduced the sculpting metaphor for volume data [GH91]. The sculpting tools within the system are represented as volume datasets in their own right, and the interaction between these tools and the volume dataset governs the modification of the volume data. For interactive feedback, the system converts the volume dataset into a surface mesh using the Marching Cubes algorithm, working only on the areas affected by the tool at each frame. Their system uses a six-degree-of-freedom input device for user input.

Wang and Kaufman [WK95] apply the sculpting metaphor to volume data. A selection of carving tools is presented to the user, who can apply them to the volume data and see the results rendered in real-time. The tools are implemented as volume objects (similarly to [GH91], which simplifies the carving operation to a CVG diff operator. The authors also give a sawing tool which uses a splatting approach to allow the user to cut out arbitrary shapes. A surface-based method called Warp Sculpting was introduced by Gain and Marais [GM05], which represents the sculpting tools as distance fields to allow for smoothly weighted operations.

The 3D Chainmail algorithm was introduced by Gibson [Gib97] and is regarded as one of the very first volume deformation algorithms. The method treats the volume dataset as a set of linked elements in a lattice. Moving one element of the lattice affects connected (in this case, 6-connected) elements by propagating the stress recursively through the lattice. The user can specify the object's elasticity properties, which affect how the stresses are propagated. The authors specify no rendering algorithm as such, rather the rendering of the deformation is a simple forward-projection of the voxels as points using OpenGL.

Westermann and Rezk-Salama [WRS01] present a freeform volume deformation technique that utilises consumer graphics hardware. The technique is based on the idea of explicitly defining the shape of the volume object to be deformed using a triangular mesh, as a preprocessing step. The authors' choice of rendering algorithm involves viewer-aligned cutting planes (discussed in Section 4.2). Cutting planes intersect with the shape geometry, and the new volume sample point is chosen based on the inverse of the shape deformation (a backward-mapping scheme). Rezk-Salama *et al.* [RSSSG01] produced a followup paper detailing a similar scheme, this time maintaining a static geometry and deforming only the texture coordinates, using the inverse of the deformation for the new coordinates calculated for the slice in the vertex shader. The geometry in this case is specified as piecewise linear patches to allow for a more accurate interpolation.

3.7.1 Free-form Deformation

Winter [Win02] gives a method for rendering parametric volumes, with a specific example of Bézier volumes. The approached discussed divides the parametric volume into a set of tetrahedra, storing the voxel scalar values at each vertex. From this parametric volume the forward-mapping is computed for each vertex to transform the volume into Euclidean space. The rendering algorithm is essentially reduced now to an irregular volume rendered – each ray through Euclidean space is tested against the tetrahedra and once a sample point within a tetrahedron is found, the new sample value is computed from the tetrahedron's vertices using barycentric interpolation.

A customised octree is used to accelerate the process, and additional optimisations are included by exploiting coherence between the rays and cells. Normal computation via central differences was found to produce artefacts due to the tetrahedral approximation of the Bézier volume, so instead the normal is computed using central differences in dataset space and then analytically mapping this normal into Euclidian space.

3.7.2 Sweeping Metaphor

Winter and Chen [WC02] introduced Image-Swept Volumes, whereby a 2D planar template is swept through a 1D trajectory to produce a swept volume. The authors give examples of sweeping templates defined as images through Bézier curve trajectories, and also simple sweeps defined by rotating the template around an axis. The method is not limited to simple 2D templates, and the authors additionally give examples of sweeping varying data such as the slices of a volume dataset through a trajectory to produce a deformed volume. A fire effect is achieved by sweeping frames of a flame video around an axis. In order to render the resulting sweeps, two methods are given: voxelisation and direct evaluation.

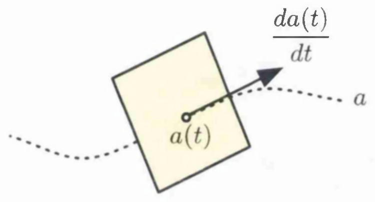


Figure 3.7: Sweeping Template (as used by Winter and Chen [WC02])

The voxelisation method places the planar template(s) along the trajectory and voxelises them into the new volume dataset using linear interpolation; the planar templates in the case of voxelisation therefore are placed using the forward-mapping of the sweep function. To achieve an accurate approximation, a recursive subdivision method is used to position the planar templates on the trajectory with smaller and smaller distances between planes until a tolerance threshold is met. The advantage of voxelising the swept volume in this manner is that the resulting volume dataset can now be rendered using a standard direct volume renderer with no modifications, and can thus be integrated into any direct volume rendering pipeline. The disadvantage however is the amount of time required to generate this volume, and the amount of additional space required to store it.

The authors additionally give a discussion on direct evaluation of the sweep without voxelisation. In order to achieve this, the inverse of the mapping function is required. An analytical breakdown of obtaining the inverse of the mapping function is given for rotational, translational, and general sweeps along a trajectory. Computation of the normals in deformed space is not computed as in Equation 3.2, the authors instead note that additional computation of Φ^{-1} would be prohibitively expensive and therefore give a more optimal solution by mapping the normal from deformed space to world space using the sweep information.

3.8 Segmentation of Volume Data

Segmentation is a vast subject, due to both the demand for segmentation methods (particularly in medical image analysis) and the complexity of segmentation in general. Segmentation for 2D discretely sampled objects (i.e. images) has been widely researched and documented; indeed simple segmentation methods exist in off-the-shelf image editing applications for defining object selections (commonly called the 'lasso' tool or the 'magic wand' tool). While great efforts have been made to devise techniques for segmenting volume datasets, unfortunately most 2D segmentation methods do not naturally lend themselves to the additional dimension. This is further exacerbated by the fact that there is a mismatch in dimensions between the 3D dataset and the 2D framebuffer, making user-interactive approaches much more difficult.

The segmentation of images can be defined as the process of partitioning the image into disjoint regions, based on some criteria specified by the user. Numerous segmentation methods have been available in computer graphics aimed at giving users the ability to delineate objects in an image; for example in the media industry where a graphic designer may wish to separate a foreground object from the background to apply different operations to each. Segmentation is therefore one of the primary methods in which semantic information can be artificially added to a discretely sampled object.

The next section looks at the motivation for segmentation, particularly for volume datasets. This is followed by a look at various segmentation techniques; firstly image-based techniques which focus on the low-level data, followed by shape-based techniques which attempt to form shapes from segmented regions. These sections also briefly discuss recent advances in GPU-based segmentation methods.

3.8.1 Motivation

The segmentation of volume data is an important operation in the medical industry. Doctors might wish to take CT/MRI datasets and remove parts obscuring the parts of interest – e.g. separating the grey and white matter in the brain is an important step in identifying many diseases and conditions of the brain, also the separation of the various regions of the brain aids the doctor in diagnosing the problem.

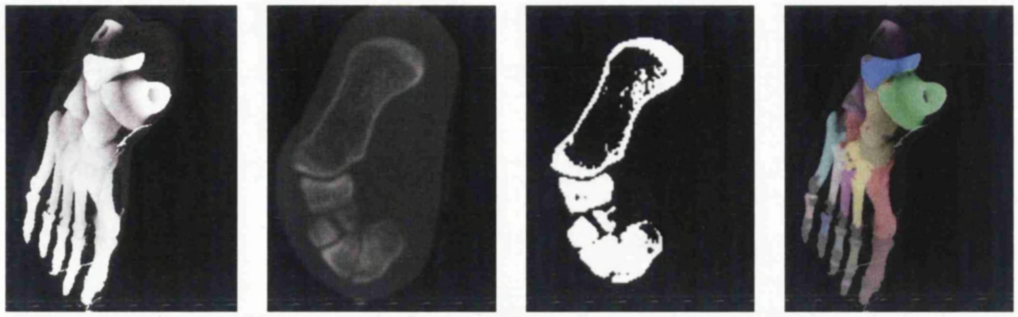
Segmentation is also a means of adding semantic information to a volume dataset. Such semantic information can be invaluable in producing deformations and animations of such datasets, particularly for character-based animation (as discussed in Section 3.6), where, for example, the arms of the Visible Human dataset need to be delineated from the body in order to move the arms freely without surrounding parts of the body moving with them. In case of the Visible Human, the arms are touching the torso, so it is clear that a reasonably high amount of control and precision would be required to achieve an accurate result.

Segmentation methods can be broadly classified into two categories: *stochastic* and *knowledge-based*. Methods belonging to the former group work only with the data available, with no knowledge of what the samples in the dataset actually represent. Methods belonging to the latter category work with some knowledge of what the data represents, and can therefore be optimised for the specific task at the expense of a less generalised algorithm.

Generalised segmentation algorithms require a larger amount of user interaction in general, prompting the user for start points for seeding algorithms, or the fine-tuning of input parameters that govern the segmentation. Indeed, quite an important and widely-asked question in computer graphics is whether there exists a segmentation algorithm that can work proficiently on a wide range of input data, with no user interaction. Such a method could employ Artificial Intelligence methods to provide a result similar to that of a human segmenting the dataset by hand.

3.8.2 Stochastic Segmentation

Segmentation methods for 2D images have been widely documented. The simplest method of segmentation is *thresholding*, whereby a threshold value t is set, and all samples above or below this value are deemed part of the final segmented object. Figure 3.8(a) shows the CT foot dataset rendered with a transfer function acting as a segmentation result. In this image, all voxels above 112 were set to a white colour with full opacity (the bone), and all voxels below were set to a reddish colour with a very small opacity (the skin). This method



(a) Threshold (b) Seed Fill before & after (c) Watershed

Figure 3.8: Automatic and semiautomatic segmentation of the CT Foot dataset by means of *(a)* a simple threshold implemented as a transfer function; *(b)* a seed fill, where the left image shows the original data and the right image shows the seed fill result; and *(c)* different parts of the CT foot segmented using the Watershed algorithm

is adequate for a small number of scenarios, but it offers little in the way of control and often produces poor quality binary-segmented results.

Algorithms can be designed to choose thresholds automatically from histogram analysis. Work presented by Kang *et al.* [KEK03] presented a method for inferring thresholds based on analysis of the histogram, locating Gaussian curves automatically. If the thresholds have a transition zone inbetween, the classification relies on the standard deviation of the 26 neighbours of the voxel. The remainder of the algorithm relies on morphological closure and region-growing techniques, with some user interaction required where, for example, there is a very fine gap between two objects.

Region-growing techniques such as the flood filling algorithm can be easily extended into 3D. A start point (the *seed*) in the dataset is chosen, and this seed point is grown – that is, voxels are added in a recursive manner until they hit a predefined boundary. Figure 3.8(b) shows (left) a slice of the CT foot dataset, and (right) the voxels captured with a seed fill. The problems with this approach are carried over from the 2D approach typically used in simple image editors. Firstly, the filling process easily spills out of the desired area if just a few boundary voxels are missing. Second, the result is highly dependent upon the user's choice of seed point. A seedless version of the method was developed by Zheng *et al.* [LJT01] for volume segmentation.

The watershed algorithm [VS91] has been applied to 2D images in an attempt to improve the situation. The watershed algorithm treats the image as a set of disjoint *basins*, divided by the edges of the object of interest. The basis are gradually filled with water, and watersheds are defined to divide areas where one basin spills into another. The watershed transform is easily extended into 3D, and hierarchical versions have been devised [HP03] that maintain the history of the basins merging in order to provide quick user feedback after the initial computation. Figure 3.8(c) shows a result from the watershed algorithm applied to the CT foot dataset. In this image, each basin has been assigned a different colour. The algorithm has made a reasonable job of segmenting the metatarsals and phalanges.

3.8.3 Using Artificial Intelligence

A recent trend in the area of image segmentation has been the use of artificial neural networks [RAA00]. The strength of using such methods is that, by definition, they can learn from past attempts on other datasets and refine themselves in future attempts. The use of artificial intelligence can be of great benefit to an application as complex as the segmentation of discretely sampled objects as it is clear that some degree of intelligence is required to infer higher-level semantic information; e.g. what exactly constitutes a bone structure in a CT dataset requires a good understanding of the intuitive notion of what a bone constitutes in terms of its shape, colour, and a good general knowledge of the body in general.

Ahmed and Farag [AF97] use a two-stage system comprising of self-organising analysis networks and self-organising feature maps for segmenting a CT dataset. Self-organising feature maps attempt to represent the three-dimensional data in only two dimensions, grouping together similar objects with no user interaction. They are also trained automatically.

LEGION (Locally Excitatory Globally Inhibitory Oscillator Network) systems go one step further into the problem of understanding how the visual cortex interprets multiple objects in a scene. The systems are derived from experimental evidence on this part of the brain, and also some theoretical work in mathematically modelling the processes. The central idea is that the oscillations in the visual cortex can be mimicked in software, attempting to reverse-engineer the brain's framework for detecting and identifying the imagery received by the eyes. Because of the complexity of such a system, simplified algorithms are developed to work with large datasets [SWY99].

3.8.4 Data-Mining Methods

Clustering can be defined as a measurement of similarity in image regions. Clustering techniques attempt to group voxels together that display similar predefined characteristics. A N-dimensional vector is built from each voxel based on the properties of that voxel, and the set of all N-dimensional vectors is then fed into the clustering algorithm. One of the most popular clustering algorithms is K-means clustering, which attempts to form n disjoint, nonempty subsets by grouping together 'similar' voxels. Closely related as fuzzy clustering, which utilises fuzzy 'if-then' rules to determine object membership.

Popular algorithms from graph theory can be used in conjunction with clustering. Edges are built between vertices (in the case of a volume dataset, voxels) that display similar properties [WL93]), and are then removed from the graph where the vertices touching the edge fail to satisfy the similarity measure. The result is a graph G that consists of n unconnected subgraphs, which correspond to the segmented regions.

Markov Random Fields are a statistical method that can be used alongside clustering to achieve automatic segmentation. Such a field stochastically defines the local properties of the dataset in a completely generalised manner by modeling spatial interaction between voxels [RGR97]. Such algorithms unfortunately are computationally expensive by nature and are heavily influenced by the initial controlling parameters, making segmentation a trial-and-error experiment.

3.8.5 User Interaction-Intensive Approaches

Human interaction is often required in medical image segmentation to fine-tune any segmentation efforts made by the system, and also to guide the algorithm as it attempts to provide segmentation regions. This approach is used with great effect in the PAVLOV system [KK99], where a parallel CPU system powers the rendering of the segmented dataset to allow for real-time updates. The user is invited to segment the dataset using thresholding and morphological dilation and erosion operations. Sherbondy *et al.* [SHN03] use a GPU-based implementation of the seed-fill algorithm to segment regions of interest. A significant speedup over that of SSE2-accelerated CPU code was achieved. Combined with hardware-accelerated rendering, the user is able to segment the dataset and view the results in real-time.

Interactive segmentation need not always use the GPU for the segmentation stage; the CPU can be used to perform the segmentation step, and the rendering feedback of the segmentation process can be offloaded to the GPU. Sherbondy *et al.* [SHN03] render the isosurface of the segmented object on the GPU. Hadwiger *et al.* [HBH03] show the possibility of providing extremely high-quality feedback of volume segmentation techniques that produce 3D masks as output; demonstrating the potential for many interactive CPU-based segmentation techniques to use the GPU to render feedback.

3.8.6 Introducing Domain-Specific Knowledge

Stochastic approaches can produces reasonable approximations of the intended result from the user's perspective, but very often, introducing a domain-specific solution can produce highly optimised results tuned for the problem at hand. The requirement of accurate and automated segmentation techniques for use particularly with medical data often prompts researchers to devise new algorithms and hybrid techniques which are optimised specifically for a subset of the human anatomy. Such algorithms can often excel completely automatically at the task at hand.

An introduction of domain-specific knowledge to a segmentation algorithm can be achieved by introducing a pre-generated atlas for the known dataset. An atlas can be considered as a pre-segmented dataset similar to the dataset to be segmented (the *target* dataset) and is used as a reference for the segmentation process. The atlas can be *registered* with the dataset to be segmented, that is, aligned to match the target dataset as closely as possible (an excellent review of medical image registration techniques can be found in [MV98]). Once the registration is complete, the correspondence can be used define which voxels belong to which object.

This approach is adopted by Straka *et al.* [SCD+03] where a watershed transform is combined with the atlas to create partitions of similar areas of the dataset. Atlas information can also be factored into previously stochastic methods, such as Snakes [KWT88, BB03] or statistically-based segmentation [FRZ+04] to further improve the results. Grau *et al.* give an improved watershed algorithm that uses prior information from an atlas to improve the accuracy [GMA+04] of the standard watershed transform.

3.8.7 Shape-Based Segmentation

While image-based approaches can result in the desired result for simple cases, they are less than adequate in many cases due to their operating on the image pixels with no knowledge of the object they are attempting to define. Shape-based approaches operate on shapes in the image, resulting in a much more intuitive segmentation.

One such example of shape-based segmentation is one of the many energy-minimising models that exit for segmentation. Energy-minimising models are explicitly defined geometric models defined within the object of interest. Forces then act upon this model to change its topology, attracting it towards the desired result. Such models used for segmentation purposes are commonly known as *deformable models*.

The most popular energy-minimisation technique for image segmentation was introduced by Terzopoulos *et al.* [KWT88]. The *Snakes* system uses energy-minimising splines (called *snakes*) defined inside the target object. Once defined by the user roughly around the object, the snake attempts to minimise its internal and external energy as:

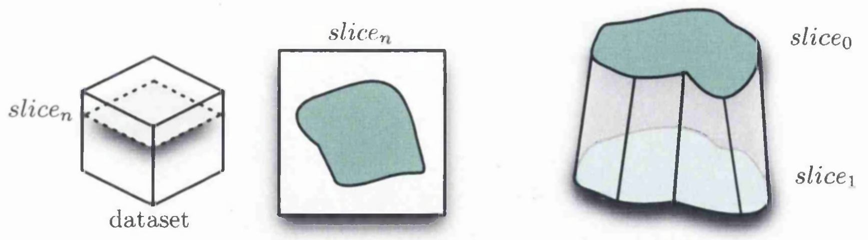
$$E_{snake} = \int_0^1 E_{int}(s) + E_{ext}(s)$$

The internal energy E_{int} is defined as the energy of the snake itself – e.g. the severity of the bends, and its total length (the amount of stretching). This energy gives some predictability for the shape of the snake and ensures that a satisfactory topology is maintained. The external energy is defined as the image energy (and any user-defined constraints specified through the user interface), and it is this energy which attracts the snake towards areas of interest in the image. The most common image energy functional would be an edge detection function, which returns smaller values near edges in the image and larger values in solid areas.

Ballerini *et al.* developed a bone age assessment system where multiple snakes are defined around bones [BB03] in order to segment the bones of a hand away from the flesh. Miller *et al.* [MBL+91] extended the energy-minimisation model into 3D. Their system works by defining a polygonal balloon inside the object of interest. Once defined, the balloon grows outwards due to its internal energy, and due to the external energy, it converges to the shape of the object. An internal energy functional additionally maintain a correct topology for the balloon, keeping it closed. The image energy functional used is quite a simple threshold operation which gave reasonable results.

3.8.8 Extending 2D Methods to 3D: Slice-Based Segmentation

Slice-based approaches to volume segmentation utilise existing 2D segmentation approaches, but extend them into 3D by operating in a 2.5D manner – building 2D slices of the data and then stitching the slices together into a 3D segmentation result. Figure 3.9 demonstrates a slice-based approach to volume deformation where *(a)* a slice of the volume dataset has been shown to the user, and a Snake defined on the slice, and *(b)* the system has constructed a 3D representation of the segmentation by connecting the slices into a boundary representation.



(a) A slice of the dataset is shown to the user (b) A mesh connecting the slices

Figure 3.9: Segmenting a dataset with 2D slices

The simplest volume segmentation techniques therefore can be implemented by presenting slices of the dataset to the user and allowing the user to segment each slice until the final slice is reached. More sophisticated methods would allow for arbitrarily-positioned 2D planes to be defined within the data, with the user shown an interpolated slice image from the plane. This process in general is known as *contour-connecting*, and has origins not in segmentation but in obtaining isosurfaces from volume data by operating on slices of the dataset and joining the slices into a final triangular mesh.

Numerous approaches exist for joining contours together. The most common approach is to discretise the contour into a series of vertices and then attempt to join the vertices between each slice; however this approach is plagued with problems such as irregular shapes, sudden large changes in the shape of the contour, and the existence of multiple contours. Fuchs *et al.* [FKU77] treat the problem as graph-theoretical by attempting to create a graph of minimal cost between slices. Shantz [Sha81] treats multiple contours in a single slice as one contour by concatenating the contours before joining – solving what is known as the *correspondance problem* where it is unclear which contours are related [Mey94]. Ekoule *et al.* [EPO91] use a heuristic approach, dividing the contour into convex and concave regions and creating intermediate contours for the cases where contours split or merge. Giertsen *et al.* [GHF90] give an interactive approach to connecting the slices. Jones and Chen [JC94a] give an approach based on creating a distance field beforehand, which is used in conjunction with the Marching Cubes algorithm to create a final triangular mesh. Bajaj [BPS96] give a method for creating an isosurface triangular mesh from a volume dataset by choosing an initial seed point and propagating the mesh around the isosurface, effectively propagating the contours.

3.8.9 Level-Sets

The energy-minimisation techniques discussed previously all maintain explicit representations of the surface at any time. *Level-set* methods represent the surface implicitly by solving partial differential equations, and are conceptually similar to the shape-based methods discussed in the previous section; except the underlying surface representation is implicit rather than an explicit surface.

A surface S can be expressed as the level set of a volume dataset as:

$$S = \{s \mid \mathbf{V}(s) = k\}$$

where k is an arbitrary isosurface. When $k = 0$, the level set is the zero level set of $\mathbf{V}$.

The advantage of an implicit representation is immediately apparent when one considers the splitting, rejoining, and generally unpredictive nature of some objects; since an explicit representation is only generated from the implicit representation when required, such topological changes are handled naturally. The actual segmentation process involves evolving the level-set surface for an initial curve defined in a slice of the dataset. An explicit surface can be extracted for rendering if required.

Whitaker *et al.* [WBMS01] give a framework for the level-set segmentation of volume datasets, giving excellent results for a complex (that is, the isosurface is complex) dendrite dataset. The initialisation of the isosurface is achieved interactively. Lefohn *et al.* [LKHW04] give a GPU implementation of a level-set solver for volume segmentation, giving interactive frame rates for the evolution of the isosurface. Cates *et al.* [CLW04] give a similar, interactive method that evaluates the surface on the GPU that relies on a novel GPU memory management scheme.

3.9 Summary

This chapter has introduced the most important and relevant concepts of animation and deformation, with a particular emphasis on their application in volume graphics. With volume graphics becoming a much more widespread and accepted field of computer graphics, research is increasingly becoming focused on the manipulation of volume datasets. However, the amount of research focused on the area is still rather limited in terms of scope; concentrating mainly on very specific applications such as surgery planning.

A look at animation and deformation was given at the beginning on the chapter, with particular emphasis on how the two concepts are linked implicitly, and the control systems that can be used for animation. The subject of deformation of volumetric data was introduced in Section 3.4, introducing the concept of Spatial Transfer Functions to the reader and giving an important overview of the low-level technical issues that exist when deforming discretely sampled objects such as volume datasets; in particular, the forward and backward mapping of volume data was discussed, and the manner in which this data must be subsequently rendered.

An review of the most important works in the field of volume deformation and animation was conducted in following sections, including a review of the applicability of skeletal systems (Section 3.6), and soft-body deformation and sculpting (Section 3.7) methods. The chapter ends with a comprehensive discussion on volumetric segmentation techniques and the manner in which segmentation can add the necessary semantic information to volume datasets to enable complex deformation and animation.

Chapter 4

GPU Volume Rendering

Contents

The past ten years have seen a revolution in the power and capabilities of consumer graphics hardware. Moore's Law, which states how the number of transistors on modern processors increases over time, has been observed to break down when considering the powerful parallel processors found on consumer Graphics Processing Units (GPUs) due to their excessive growth. *Consumer* in this context implies that the hardware can be purchased inexpensively at a local computer store, and such hardware is most commonly utilised to play modern computer games.

Due to the increasing power of the hardware and ease of use of the APIs, many researchers have been using graphics hardware for purely computational tasks – something unforeseen by GPU manufacturers initially, although NVIDIA have been quick to respond with CUDA (Compute Unified Device Architecture) which attempts to bridge the gap between the CPU and GPU by providing both on one chip [CUD].

Volume rendering is often referred to as an 'embarrassingly parallel' problem – the primary method of rendering (raycasting) can be computed entirely in parallel since there is no inter-ray dependence unless this dependence is introduced explicitly by a specialised algorithm. Such a rendering algorithm is perfectly suited to the GPU, which can execute many of these rays in batches. Such parallelism can be defined as:

> *[A problem of size N, where] it is quite easy to achieve a computational speedup of N without any interprocess communication.... each process is given $\frac{1}{N}$ of the computations that can be independently done, and the results do not need to be combined in any way.* [Har]

This chapter therefore gives an overview of the tools, technologies and algorithms used to produce real-time volume rendered images using the GPU; beginning with an overview of the OpenGL pipeline and the parts of the pipeline exploited for volume rendering. This is followed by an overview of GPU volume rendering techniques; including a review of important forward-projection rendering techniques implemented on the GPU and their implementation issues.

The latter half of the chapter documents an example implementation of a new hybrid CPU/GPU raytracer that takes advantage of the GPU for computing ray/isosurface intersections within a volume dataset. A raytracing software application, Igneus [Spe] was used in the development process. The Igneus raytracer provides realistic images using global illumination, and is extensible via an object-oriented interface. A customised shading class was implemented for the hybrid functionality.

4.1 The OpenGL Pipeline

The OpenGL API is a popular graphics API in the research industry due to its extensibility and cross-platform implementations [WNDO99]. Any functionality that does not already exist in the base API can be added by hardware vendors via the extension interface, making OpenGL attractive for algorithm developers wishing to take advantage of the very latest hardware capabilities.

Figure 4.1 gives a simplified version of the OpenGL graphics pipeline. The vertices of the primitives specified by the user are sent to the vertex unit. The standard operation in this unit is to modify the vertex position to bring it from world space to view space. Next, the vertices are sent to the rasteriser. The rasterisation process converts the primitives specified by the incoming vertices (usually a sequence of triangles) into screen-space fragments. A fragment can be considered to be a pixel in view space corresponding to a pixel in the framebuffer, except a fragment has an additional depth value specifying its z-position in view space.

A recent addition to graphics hardware has been the ability to fetch texture data in the vertex shader – referred to as *vertex texture fetch* functionality. This functionality allows for large amounts of data to be input to the vertex stage encoded into the colour values of a 2D texture. Before this capability, data sent to the vertex shader was only accessible via arrays that had to be indexed at compile-time.

The role of the vertex processor is to take each vertex comprising the object to be rendered and bring it into view-space ready for the fragment shader; achieved by multiplying the vertex's position in world space by the current model-view-projection matrix. The vertex can potentially have many associated attributes specified by the API, such as colour / opacity, texture coordinates, and so on.

Before a fragment is processed by the fragment shader, it must survive several tests to determine whether it will contribute to the final image. The most well-known and commonly employed of these is the *depth test*. In real life, opaque objects closest to the viewer obscure objects behind them. To simulate this, the graphics pipeline includes a *depth buffer* (otherwise known as the z-buffer), which contains a scalar value for each pixel in the framebuffer

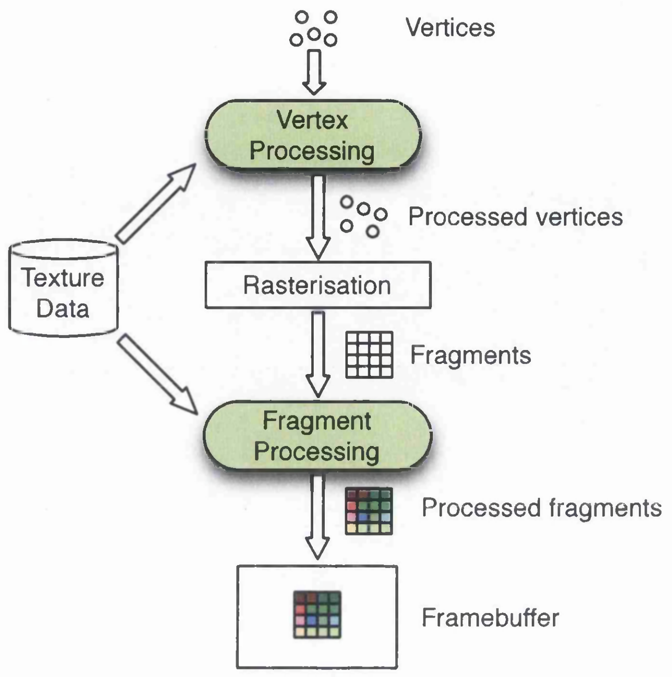


Figure 4.1: The OpenGL Graphics Pipeline

denoting the depth of the closest previously-processed fragment. The depth test compares an incoming fragment's depth to the depth stored in the depth buffer. If it is less, then the fragment survives, otherwise the fragment is discarded as it will not contribute to the final image. The depth testing procedure is customisable for different applications however, and can be easily modified in OpenGL using the following three functions:

- *glEnable/glDisable with bitmask GL_DEPTH_TEST* – enables/disables depth testing
- *glDepthFunc* – changes the depth comparison function. Defaults to $<$, but can be changed to other comparisons such as $>$,$=$, etc.
- *glDepthMask with GL_TRUE or GL_FALSE* – enables/disables writing to the depth buffer, but does not affect depth buffer testing.

The *stencil buffer* provides the user with a means of fragment culling in a similar manner to the depth buffer, but in a much more arbitrary manner. The *stencil test* is performed before the depth test, and tests the current fragment against its competitor in the stencil buffer using a function defined by the user. The stencil buffer is typically used to mask out areas of the screen to avoid computation taking place in those areas.

4.1.1 The Programmable Pipeline

As the power of the graphics hardware increased, so did the demand for the realism found in other media such as film. This realism is achieved by customisation of parts of the graphics pipeline, instructing the pipeline, for example, exactly how to shade a particular pixel accurately based on its material properties and lighting conditions. A program that modifies part of the pipeline in this way is called a *shader*. An example is Pixar's RenderMan shading language [Pix05], which is used in Pixar's motion pictures. The shading instructions in this case are evaluated on the CPU, not using graphics hardware, and are thus simpler to implement.

Developers had previously achieved complex effects by using multipass techniques [POAU00]. Due to the complexity of designing such effects, developers often created their own shading language that translates into C/C++ API calls, such as the Quake 3 Shading Language employed in the computer game Quake III : Arena [JH99]. In 2001, NVidia released the Geforce 256, the first consumer programmable graphics card. The card allowed for developers to program the pipeline in a well-defined manner by employing shaders to alter to the vertex and fragment streams.

The shaders written for the Geforce 256 were written in the card's native assembly language, and were thus neither portable nor easy to write, maintain and understand. Cg ('C for graphics') was introduced by NVIDIA in 2003 [MGAK03] to solve these problems. Cg provides developers with a high-level language for developing shaders, improving program portability and readability. The language has a C-like syntax, but differs from C in several important ways. The introduction of vector and matrix types as a primitive type allows for much cleaner code by allowing the developer to write operations that act on all vector components in one statement, without worrying about the messy array handling caveats of C.

Microsoft have also produced their own high-level language called HLSL (High-Level Shading Language), with almost identical functionality to Cg as both generate native code for the same market of consumer GPUs.

The programs that run in the vertex and fragment stages are called *shaders*. Shaders running in the vertex unit can modify vertex attributes such as its colour/opacity, texture coordinate, etc. The most usual operation is to modify the vertex's position to bring it into clip-space ready for subsequent rasterisation operations. Vertices cannot be destroyed explicitly, but can be removed from view and thus from rasterisation by setting their output positions to fall outside of clip-space. Fragment shaders work on one fragment at a time, and can either output a final fragment colour and opacity, or can destroy a fragment altogether. Fragment programs can additionally modify the depth of a fragment.

4.1.2 General-Purpose Computation using Graphics Processing Units

General-Purpose Computation using Graphics Processing Units (commonly abbreviated to *GPGPU* by its community) is a relatively new area of research in the field of computer science. The area is driven by two factors – firstly, the ever-increasing power and parallel nature of GPUs, and secondly, the increasingly flexible manner in which these units can be programmed due to the increasing abilities of the fragment shaders (dynamic branching and loops) and the manner in which they are programmed using higher-level languages. The Geforce 6800 is capable of performing hundreds of gigaflops (for 32bit IEEE).

In the same way that CPUs improve their architecture and increase the number of transistors, GPUs follow a similar pattern. The clock speeds of the chips found on modern GPUs continues to increase, and as memory technology improves and reduces in price, memory bandwidth increases and latency decreases; however it has been observed that bandwidth is increasing at such a rate that latency is becoming the predominant factor in computations [Owe05, Har05].

The GPU by nature is well-suited to highly parallel algorithms, due to the non-dependent nature in which it operates on vertices and fragments – each computation (pixel) consists of an input (texture data) and output (final pixel colour) that is non-dependent on all other computations (pixels) occurring in the current job. In comparison to the CPU, however, the GPU's ability to perform control-flow operations is very poor as they were simply not designed for such instructions – they are available for added flexibility. The number of instructions executed available per program has risen with ShaderModel 3.0 to 2^{16}, due to the introduction of loops providing the developer with an easier means of high amounts of computation.

4.1.3 The Stream Model

Typically, the stream data is encoded into a 2D texture and rendered onto a quad in front of the viewport, ensuring a 1:1 correspondence between texture texels and framebuffer pixels. The kernel is the fragment program operating on the incoming texels – only one fragment program can be bound per pass. The fragment program is able to access the texture data, perform a computation, and write the result to the framebuffer encoded as the $< R, G, B, A >$ output. Once complete, these results are read back using $glReadPixels$, or the equivalent. This is the most common method of operating on stream data, though it is also common to utilise the vertex shader for intermediate computations, with results being interpolated across the face of the quad.

Though effective, this system is often difficult to work with due to differences between GPU capabilities, driver issues, and the amount of up-to-date knowledge required by the programmer. Projects such as Brook [BFH+04] and Sh [Lib] have attempted to alleviate this problem by providing some degree of abstraction in the form of either a new language or an API.

4.1.4 Render Targets

Pixel Buffers (PBuffers) were finalised into the OpenGL specification in 2000, and provided developers with an offscreen buffer into which they could render intermediate data. An additional extension called *render-to-texture* enabled developers to copy the pbuffer data to a texture. This allows for a technique that is now become known as *ping-ponging* – where two textures are utilised, one acting as the input texture, and one as the render target. The roles of these textures are switched between passes, allowing for multiple computational passes. While pixel buffers were quickly adopted for such use, their disadvantages made them impractical and difficult to work with – their implementations were entirely platform dependent, and switching between buffers required an OpenGL context switch.

Framebuffer Objects (FBOs) solved these problems by providing a platform-independent solution that was simpler to use for the developer, enabling the attachment of either off-screen render targets or textures to the bound FBO. Multiple Render Targets (MRTs) were introduced in the ShaderModel 3 specification, and allowed for writing to four render targets in any one pass. The ability to store unclamped (i.e. not clamped into the $[0, 1]$ range) full 16 or 32-bit floating point data in textures was introduced in the form of the *float_buffer* extension, driven mainly by the desire to enable High-Dynamic Range (HDR) rendering. It was quickly adopted by the GPGPU community as a means of storing full precision data.

4.1.5 Geometry Shaders

Geometry Shaders were introduced into Microsoft's DirectX 10 API, and give increased flexibility to the developer for vertex manipulation. Geometry shaders are introduced as a new element in the graphics pipeline, occurring just after the vertex shader. They are not only able to create new primitives on-the-fly, but are also able to access all of the vertices of the primitive being operated on. Due to the fact that geometry shaders are a relatively new concept at the time of writing and were introduced late into the research conducted in this thesis, they will not be utilised.

4.2 Raycasting Volume Rendering on Graphics Hardware

Volume rendering on texture-mapping graphics hardware has long been a subject of research [CN94, CCF94, DKC+98]. Volume rendering is one of a class of problems that lends itself particularly well to the GPU as it can be easily parallelised – a so-called *embarrassingly parallel* problem.

The first volume renderers to utilise standard consumer graphics hardware used the only texture mapping functionality available at the time: 2D texture mapping. The idea was to define a series of quad primitives along an axis of the data, with each slice of the dataset assigned to a quad [RSEB+00]. If the blending operation is set correctly in the graphics API, the slices are blended together in a back-to-front manner such that the volume rendering integral is discretely approximated as with raycasting methods. The system requires that the slice geometry and axis be modified to suit the current view parameters, and can produce artefacts when the slice geometry is suddenly changed as the user rotates around the dataset.

Hardware-accelerated 3D texture mapping was first discussed by Cullip and Neumann [CN94]. The paper describes an scheme where image-aligned polygon slices are rasterised inside the volume dataset. The texture coordinates for each fragment are correctly specified in OpenGL, and the density information is used as both the grey-level colour and α value to blend the slices together. The system achieved around 10FPS on a Silicon Graphics RealityEngine – impressive performance for 1994. Gelder *et al.* [GK96] implemented a similar system on a RealityEngine II two years later. Cabral *et al.* [CCF94] further extended this idea by discussing the coupling of the medical (tomographic) reconstruction process with volume rendering for CT datasets.

3D texturing capabilities were introduced by NVIDIA in 2001 with the introduction of the GeForce 3 architecture. This functionality allowed for loading volume datasets into GPU memory, with hardware trilinear interpolation. In 2002, Purcell *et al.* proposed that the stream-based computational model of the GPU was particularly well-suited to raytracing applications [PBMH02], due to the parallel and repetitive nature of the task. The authors implemented a ray-triangle intersection algorithm for the fragment shader with an estimated 54 million triangle intersections per second, almost three times the speed of a compared software raytracer.

High-quality renderings of volume data on the GPU are not, however, restricted to raycasting algorithms. Kaufman *et al.* [HQK05] give an object-order algorithm for rendering large datasets such as the full Visible Human dataset. The algorithm works by dividing the dataset into cells and projecting the nonempty cells to the screen using an octree to guarantee back-to-front traversal for compositing.

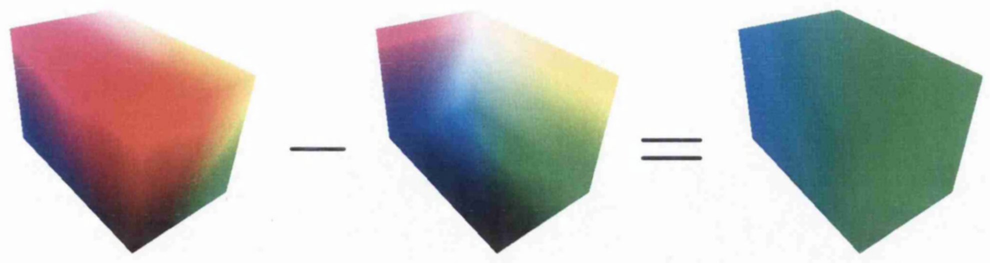
The iterative capabilities of ShaderModel 3 hardware (looping and dynamic branching) allowed for the implementation of the well-established raycasting volume rendering method [Lev88] on the GPU to be run entirely in the fragment shader with one pass [SSKE05b, MJ05]. For the application of ray marching for volume rendering, a simple involves a quad defined as the view plane. Once the quad's fragments are rasterised, the fragment shader casts rays from the fragment into the scene. The volume dataset is sampled using 3D texture lookups, with the card providing trilinear interpolation of samples. Post-integration of the density information can achieved quickly using a prepared 1D or 2D texture as a lookup function. A more elegant solution is discussed in the next section.

4.2.1 Rasterisation-based Ray Setup

Several important optimisations for the raycasting process were devised by Kruger and Westermann [KW03]. Their scheme exploits the extremely efficient rasterisation capabilities of graphics hardware to perform a ray setup stage, whereby each ray's volume intersection points are precomputed before the raycasting stage. This method requires the ability to write intermediate results to a texture (known as *render-to-texture* functionality), and therefore requires a reasonably modern graphics card. This ray data can be read back from the textures in the fragment shader and used to fire the rays through the volume data.

The objective of this ray setup stage is to obtain two 2D framebuffer-sized textures representing the framebuffer output from the operations that take place. Each texel corresponds exactly to one pixel in the framebuffer. Texture 1 represents the ray entry points into the volume, and texture 2 represents the ray exit points from the volume. These values are encoded into the RGB values of the framebuffer and consequently the texture, and are computed using a combination of vertex and fragment programs running on the GPU.

A cube (six quads in OpenGL) is defined as the boundary of the volume dataset. Figure 4.2 shows the cube rasterisation process. The front faces of the cube are first rasterised (Figure 4.2(a)), with the incoming texture coordinates for each fragment written to ray start texture's RGB colour values. This can be expressed in Cg simply as:



(a) Front faces (b) Back faces (c) Ray Directions

Figure 4.2: RGB Representation of Texture Coordinates

Listing 4.1: Fragment Shader: Outputting a texture coordinate to the framebuffer

```
float4 PassOne(
   float4 texCoord : TEXCOORD0,   // current texture coordinate
   ) : COLOR
{
   return texCoord;
}
```

Next, the back faces of the cube are rasterised (Figure 4.2(b)). The ray entry point texture from the previous stage is used as input to compute the normalised ray direction. This time, a fragment program computes the normalised direction vector of the ray based on the ray entry point (from pass one's texture) and the current texture coordinate. It also computes the distance between the ray entry point and ray exit point. This pass can be expressed in Cg as:

Listing 4.2: Fragment Shader: Calculation of ray data

```
float4 PassTwo(
   float4 texCoord : TEXCOORD0,      // current texture coordinate
   uniform samplerRECT rayEntryTex, // ray entry texture from pass one
   float3 pos : WPOS                 // window coordinate of fragment
   ) : COLOR
{
   // get ray start point from previous pass
   float3 rayStart = texRECT(rayEntryTex,pos.xy);
   // compute ray direction from above and current texture coord
   float3 rayDir = normalize(rayStart - texCoord);
   // compute ray length
   float rayLength = distance(rayStart,texCoord);
   // <r,g,b> : ray direction, <a> : ray length
   return float4(rayDir,rayLength);
}
```

Figure 4.2(c) shows the final texture (α, storing the ray length, has been set to 1 for clarity). In this case, most rays are generally heading towards positive Y, with a gradual fade to Z on the left caused by the perspective projection.

Raycasting Steps

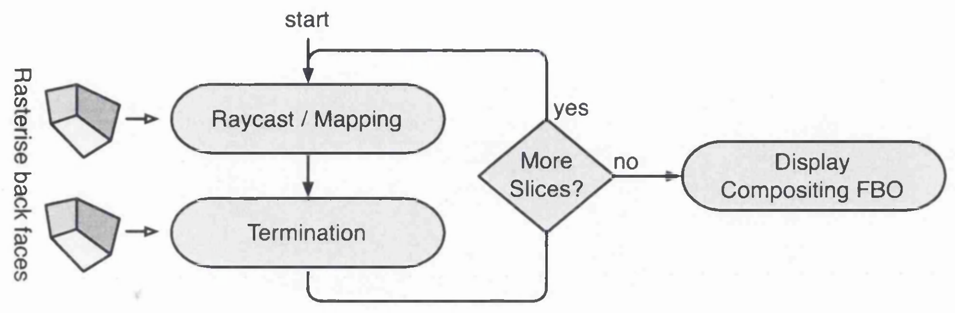


Figure 4.3: The iterative raycasting process

Once the ray data has been rendered into the FBOs, the raycasting stage can begin. The overall process is shown in Figure 4.3, which shows a high-level view of the CPU side of the scheme. The GPU code is contained within the two rounded boxes.

In order to instantiate the GPU fragment shaders, the back faces of the volume bounding geometry are rasterised in a CPU loop. The central concept is that rays are fired in small steps; for each CPU loop iteration, each ray steps *STEP_SIZE* through the world.

The code listing below shows the GPU code for raycasting. For each rasterised fragment in the current iteration, they ray data is read from the ray data textures created in the ray step stage. Given the current loop index *step* from the CPU, the ray's current position can be calculated. The code omits the compositing operations and associated variables for clarity.

Listing 4.3: Fragment Shader: Initialising and firing a ray

```
float4 Raycast(
   float4 texCoord : TEXCOORD0,      // current texture coordinate
   uniform samplerRECT rayEntryTex, // ray entry texture from pass one
   uniform samplerRECT rayDirTex,   // ray direction texture from pass two
   uniform float step,              // ray step from CPU loop
   float3 pos : WPOS                // window coordinate of fragment
   ) : COLOR
{
   // get ray start point from previous pass
   float3 rayStart = texRECT(rayEntryTex,pos.xy);
   // get ray direction from
   float3 rayDir = texRECT(rayEntryDir,pos.xy);
   // compute the ray start based on the step
   float3 rayPos = rayStart + (rayDir * step * STEP_SIZE);

   for(int i=0;i<STEP_SIZE;i++) {
      // read from volume, accumulate colour/opacity here
      rayPos += rayDir;
   }
}
```

The above listing is the first component of the raycasting algorithm. The second component (not listed) is a shader called the *termination shader*, which checks the current ray and decides whether it needs to be terminated based on its progress (i.e. has the ray exited the volume) and also some early ray termination condition. If the ray needs to be terminated, then this program writes the fragment depth to ensure that subsequent fragments for this ray/pixel always fail. It is this reason that the two programs are separate; if the raycasting program wrote a new depth value each time, then the GPU would be forced to run it every time because it could not perform early z-culling. Exploiting early z-culling is the key to many optimisations in hardware volume rendering and GPGPU techniques.

4.3 Cell-Projection & Visibility Sorting

The Z-buffer is an extremely efficient method of removing hidden surfaces on modern graphics cards. However, the Z-buffer has one fundamental disadvantage that prohibits its exclusive use even in modern games – transparencies. The blending of semitransparent materials in image-space is dependent on the order in which they are drawn. Visibility sorting is an important issue with cell-projection volume renderers, as the order of the projected cells is vital in achieving correct blending in image space; if the primitives can be drawn back to front by sorting on the CPU, then the blending of the volume data is reduced to the hardware compositing operation specified by the API.

An early algorithm was given by Newell *et al.* [NNS72] as a solution to the visible surface determination problem, in which primitives are broken down into smaller pieces until the conflict no longer exists. The A-buffer was introduced by Carpenter [Car84] and is discussed by Levoy [LW85] as a method of blending point primitives. The A-buffer does not require objects to be rasterised in any particular order; instead it maintains a linked list of fragments rasterised into each pixel.

Wittenbrink [Wit01] provides a complete design and software implementation for a hardware-acclerated A-buffer, called the *R-buffer*, and argues that the lack of such a buffer in modern hardware is detrimental both performance and ease of algorithm development. Visibility sorting must be performed in object space for rendering algorithms such as cell-projection volume renderers; this process takes not only substantial effort from the developer, but places a significant burden on the CPU – the opposite of what graphics hardware manufacturers attempt to achieve with each release. Wittenbrink *et al.* [KWW01] later develop this idea further by implementing a tetrahedral volume renderer using a software implementation of the R-buffer, removing the burden of performing a visibility sort on the tetrahedra. CPU-based sorting algorithms can be combined with GPU refinement, as with the K-buffer [CC05], which provides fragment-level refinement after a CPU-based sort.

4.4 Point-Based Rendering

Point-based graphics is a relatively new area in the field of computer graphics. An initial paper by Levoy and Whitted [LW85] suggested the use of points as primitives, arguing that as scene complexity increases, the advantages of surface mesh-based rendering begin to fade. They argue that the addition of new primitives requires new customised rendering techniques for each rendering algorithm; the research into these algorithms could be better spent elsewhere. The authors spend time detailing a rendering algorithm for such data, but neglect to discuss any possible modelling techniques. Point-based modelling systems such as the QSplat [RL00] allow for limited surface deformations and normal perturbations.

In addition to the points made by Levoy and Whitted, there are many other incentives in using point-based primitives. One such point is made by Reuter *et al.* [RTSD03] in their paper on point-based modelling and rendering via the use of radial basis functions. The authors argue that as scene complexity rises, there is a reduced need for the rasterisation of primitives. For example, for complex scenes, all vertices of a particular triangle may lie within the same pixel boundaries. In these cases, rasterisation is redundant, but unfortunately is instead replaced with a new problem unique to point-based rendering, referred to in the community as *hole-filling*.

The area has seen recent growth due to systems such as QSplat [RL00] which has an efficient software renderer and Point Shop 3D [ZPKG02] which allows for real-time manipulation of the point data. Methods for rendering point-based datasets need to account for the gaps between neighbouring samples to construct a continuous surface in image space – surface splatting [ZPvBG01b] renders object-space ellipses, with the overlap between ellipses closing the holes between samples.

4.4.1 Point Primitives on the GPU

Points are a fundamental primitive in graphics APIs such as OpenGL and Direct3D, and consequently graphics hardware are capable of integrating them into the pipeline to a degree. Until the inclusion of the the point parameters extension [Ope01] in the OpenGL specification, developers were limited to allowing the API to rasterise the points, with only colour and size available as attributes. The point parameters extension allowed for customisation of the distance attenuation of points, and was later promoted to an official OpenGL specification.

In OpenGL, point primitives can be specified between `glBegin` / `glEnd` pairs, or more efficiently using *vertex arrays*. Vertex arrays allow the developer to point OpenGL at a buffer containing a possibly large amount of vertex data and have OpenGL treat it as a sequence of vertices making up primitives. This tends to be more efficient than calling `glVertex` over and over as the function call overhead is eliminated. A more efficient system still is to use one or more Vertex Buffer Objects (VBOs) [Cor03]. The VBO mechanism allows for efficient upload of large quantities of primitive data to the graphics hardware. Unlike using the standard vertex array scheme where the data is still stored on the client side in CPU memory, the data in a VBO is held on the server side in the low-latency and high-bandwidth GPU memory. The general mechanism in OpenGL is as follows.

- The buffer is bound to the current context
- Floating-point data is uploaded to the bound buffer
- The data is rendered

4.4.2 Point Sprites

A *point sprite* [Ope01] is defined as an image-space parameterisation of a point primitive in the graphics pipeline. Point primitives travel through the vertex stages as before, except when point sprite rendering is enabled, each point is broken into a set of fragments covering the image-space bounding box of the point. An additional parameter can be set to have texture coordinates automatically generated for each fragment, giving the developer the ability to achieve complex effects such as texture-mapping of individual points or simple shaping of points.

The ability to obtain image-space texture coordinates for points allows for the correction of splat shape to be computed entirely in the fragment shader based on the point's normal relative to the viewing parameters.

4.4.3 Visibility Splatting

Point-based rendering has been implemented on the GPU using many different data encoding and manipulation techniques [BK03, KB04, BHZK05, NM05]. Modern GPUs are very well suited to dealing with the large amount of vertices and repetitive, independent processing of these vertices. Currently the most efficient method of storing the samples (points/splats) in GPU memory is to use one or more vertex buffer objects (VBOs) [Cor03], which allow for large streams of vertex data, color data, normal data, etc., to be uploaded and held in the graphics card's memory for quick retrieval.

Most GPU point-based techniques use a two or three-pass approach to correctly blend overlapping splats [CH02], as current cards do not offer α-buffer functionality. Multipass approaches sometimes write intermediate data to a texture and then use this texture data in a subsequent fragment pass to complete any additional calculations. The *visibility splatting* approach was first used by Levoy and Rusinkiewicz [RL00] for their CPU-based QSplat algorithm, though the term itself was coined later by various researchers including Pfister *et al.* [PZvBG00]. Several improvements have been made to GPU splatting since, including perspective-accurate splatting by Zwicker *et al.* [ZRB+04].

With a visibility splatting approach, a first pass generates a correct z-buffer for the current view. In order to blend visible overlapping splats, the second pass shifts the viewpoint back slightly such that the only splats now passing the z-test are those at the front of the view for each pixel. This technique has the added advantage of allowing the GPU to cull any fragments failing the depth test before entering the fragment program. The shape of the splat can be corrected in the fragment shader by using the point sprite extension and discarding fragments that lie outside of the shape of the splat; the shape of the splat can be computed from the computed image-space normal.

4.5 A Hybrid CPU/GPU Renderer

The subject of *Volume Graphics* has developed from the standard techniques of volume visualisation to an area investigating the rendering, modelling, manipulation and processing of volume objects. A wide variety of techniques have been proposed for rendering volume data – ray casting methods (volume visualization), splatting techniques, display of intermediate geometry (isosurfacing) and ray tracing. Each of these techniques has benefited from the advancement of GPU technology, most recently with the standard ray casting approach being implemented as a programmable shader using a single pass and able to operate in real-time [SSKE05a].

With advances in CPU speed, and the increasing use of PC clusters, more costly techniques such as global illumination (GI) are being considered for rendering volume data. The motivation for using GI is that it allows more complex materials to be used during rendering, it provides better visual cues for understanding data and objects, and it provides the photo-realistic quality for which graphics researchers are striving.

During ray tracing, GI requires that for each intersection point, the direct contribution from the n light sources is calculated by casting n shadow rays. In addition to a transparency ray, the specular contribution from reflecting surfaces, and the diffuse contribution from reflecting surfaces must be calculated. For example the diffuse contribution may be approximated by tracing many rays (e.g. 200) according to a Monte Carlo sampling of the projected hemisphere at the intersection point. It is this large cost, combined with the large size of volume data sets, that has meant that GI has only recently begun to be used for volume data.

In this section, GI techniques with volume data are implemented. Unlike previous methods, geometry and volume data are mixed, and materials with complex BRDFs on volume data are utilised. The role of the GPU is to take rays from the raytracing software (either primary or secondary) and provide volume isosurface intersection computations quicker than the raytracing software could itself compute them. The advantage in rendering speed comes from two aspects – firstly, the GPU is highly optimised for bulk parallel computations. Secondly, the work can be load balanced between the CPU and GPU; while the CPU is busy with lighting computations or computing intersections with the surface-based geometry in the scene, the GPU is busy computing intersections for its rays with the volume isosurface. To compare this advantage in rendering speed, a software-only method is compared to the GPU-accelerated method.

4.5.1 Related Work

First, global illumination in the area of volume visualisation is reviewed.

Beason et al. [BGB$^+$06], pre-compute a 3D texture containing illumination data for all isosurfaces within the volume. This involves extracting many isosurfaces and calculating the radiance at each vertex in the direction of its normal. The illuminance at each voxel within the 3D texture is then interpolated from scattered neighbouring vertices. For display, isosurface extraction is used to create a polygonal model, the lighting for which is determined by interpolation from the 3D texture. Although pre-computed, dynamic caustics and soft shadows on the floor are created by storing a vector of pre-computed lighting at each vertex on the floor for each isovalue. This fixes the scene to the one used during computation (e.g. a Cornell box). Their method applies to diffuse objects.

Wyman et al. [WPSH06], use a similar idea, wherein they use a pathtracer (the real-time ray tracer [PPL$^+$99]) to calculate the irradiance for each voxel using Monte Carlo pathtracing against the isovalue at the voxel (for diffuse objects). They extend the method to enable dynamic lighting on diffuse objects from a distant lighting environment (e.g. a spherical light map) by calculating Spherical Harmonics coefficients. They also suggest that non-diffuse materials could also be pre-computed and stored as Spherical Harmonics coefficients.

Both methods are characterized by very large pre-computational times on clusters – e.g. hours using 30 processors. Wyman et al. [WPSH06] suggested that it may be possible to use GPUs to accelerate the illumination computation. In fact, the work we present here on load balancing global illumination calculation on volume data provides such a method, and therefore the techniques suggested in this paper will be a valuable contribution to such pre-computation techniques by making the pre-computation phase more feasible.

Max [Max95] discusses optical models for volume rendering, including scattering, multiple scattering and shadows. Kajiya and von Herzen [KH84] propose a two stage algorithm for firstly calculating the illuminance at each voxel, and then rendering a view of the volume. Aspects of these methods can be combined into a hardware implementation [DK00, KPH$^+$03]. Kniss et al. [KPH$^+$03], give an interactive illumination model. They render (in GPU hardware) each slice of the volume data to both the view image, and a light image for calculation of direct illumination. Scattering is implemented using a (spherical) phase function (the volume equivalent of a hemispherical BRDF), which is encoded as a 1D texture look-up. As each slice is rendered, light is scattered forwards to the next slice, resulting in an approximation to indirect lighting. The key element of their approach is that the slice by slice processing is suited to GPU implementation and results in interactive frame rates.

The past five years have seen a revolution in the power and customisability of graphics hardware. Starting with the introduction of the Geforce 256, developers have been able to run programs on the graphics hardware to modify the state of vertices and fragments as they travel through the pipeline. Higher level shading languages such as NVidia's Cg, Microsoft's HLSL and the OpenGL Shading Language were introduced to allow for greater portability and ease of use. Since this programmability was introduced,research has been focused on exploiting the power, memory bandwidth and parallel nature of the GPU to perform computational tasks unrelated to graphics. Developers are now starting to use the graphics hardware not only to produce the final image, but also to produce intermediate results that can be computed quicker on the GPU than with the CPU (as with the work reviewed next). In each case, the increase in computational power from the GPU must offset the cost required to upload and download the data from the graphics hardware for there to be any benefit.

There has been some previous work on using GPUs for ray tracing. Purcell et al. [PBMH02] suggest that programmable GPUs *could* be employed for ray tracing, and evaluate various algorithms, and simulate methods in which they could be achieved. They restrict their exploration to ray tracing (with shadow, reflection and transparency rays), and path tracing with 1 ray per sample (and diffuse objects). Stegmaier et al. [SSKE05a] demonstrates ray tracing of volume data using a fragment shader where individual effects such as shadows, refraction and reflection are created by altering the path of the casted ray accordingly. The method we present here differs from these methods as it utilizes Monte Carlo pathtracing with many rays per sample. Carr et al. [CHH02] implement a ray-triangle intersection algorithm on the GPU, and exploit ray caches in a similar way to those presented here to optimise the GPU utilization.

Barsi et al. [BSKS05] implement GI on the GPU. They calculate visibility by rendering the scene from one point (the shooter), and check the depth distances to all other points (the receivers) to see if they are occluded. The points are the centres of texels mapped on the surface. The radiance at the receivers is updated at each iteration. Shooters are chosen randomly, and convergence is achieved by averaging each image with all previous images. Photon mapping is implemented on the GPU by Purcell et al. [PDC$^+$03] using two methods. The faster method limits the number of photons per grid cell, and employs the stencil buffer to ensure the number of photons is incremented correctly. Carr et al. [CHH03] implemented radiosity on the GPU using a Jacobi iteration (because it is suited to parallel stream processing), however they found that the GPU implementation was slower than the CPU solver, and restricted the number of tiles they could use (to 2048).

Here, we are not attempting to make any approximations in order to achieve a real-time algorithm and thereby sacrifice quality, but rather we are employing high quality and costly GI techniques. We are using a backward sampling for diffuse and specular reflections, and so the method presented here is more comparable to the gathering methods [BGB$^+$06, WPSH06], than the latter shooting methods which are known to converge faster, but do have some visual problems. The main contributions over previous work is that we attempt to load balance the work across the CPU and GPU, we mix geometry with volume data, and we integrate a volume animation system. Our approach would also reduce the pre-computation times of GI 3D textures for visualization of isosurfaces.

4.5.2 Hybrid Method

The work presented in this section integrates several pieces of software to create one system for the GI rendering of volumes and surfaces. For volume data, an existing fast software isosurface ray tracer Igneus [Spe] (e.g. 5fps for the CThead at 300 $\times$ 300 on a 3GHz P4) and a GPU ray caster which has the possibility to calculate deformations on the GPU. Both renderers are able to return intersection points with the volume, and surface normals. It is the integration of both these renderers that allows the comparison of CPU-only rendering with load balanced CPU/GPU rendering.

The Igneus ray tracer also handles complex surface shaders (BRDFs and BSSRDFs), photon mapping, irradiance caching, etc. Figure 4.4 demonstrates a globally illuminated CThead with a glossy Phong shader simulating a gold material. Figure 4.5 demonstrates the use of different materials for different isosurfaces within the volume.

The software isosurface ray tracer is integrated seamlessly into the object-oriented development model of the ray tracer. Implicit, parametric and fractal surfaces are handled by

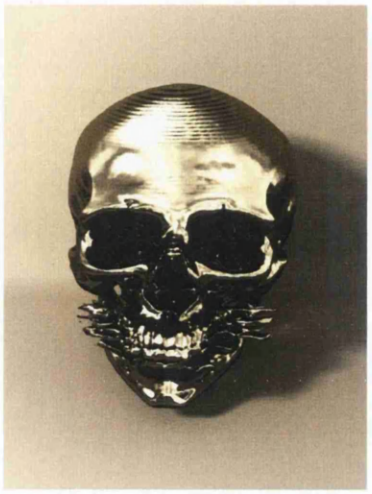


Figure 4.4: A glossy Phong laminate BRDF is applied to the CT Head Dataset

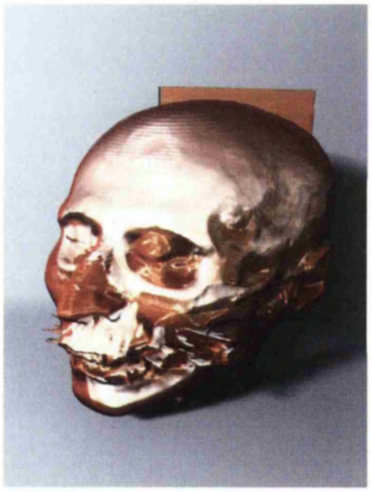


Figure 4.5: Material transfer functions - the skull is rendered with a diffuse iridescent painter and the skin with a translucent shader overlaid with a diffuse, iridescent painter and a glossy Phong laminate

```
Initialise outgoing packet with n primary rays
Cache layer = 0
While outgoing packet size > 0
        Cast rays using GPU
        Cast rays using CPU and update if surface is nearer
        Store intersection data in cache layer element
        Swap outgoing and incoming packets
        For each ray in incoming packet
                Run intersected surface's illumination model
                Store spawned rays into outgoing packet
        End For
        Increment cache layer
End While
```

Figure 4.6: Pseudo-code for ray caching.

their respective intersection classes, and this paradigm is simply extended to handle volumes. This direct compatibility makes the shading and illumination algorithms already implemented within the ray tracer (photon mapping [Jen04], irradiance caching [WRC88], complex BRDFs and BSSRDFs [JMLH01, JB02], etc.) automatically available to use with volume objects.

The load balanced approach introduced in this section makes use of programmable graphics hardware by offloading volume intersection operations onto the GPU. However, the parallel processing capabilities offered by this approach requires that rays be cast in substantial bundles in order to make full use of the hardware and to avoid penalties from the initialisation overhead. This mode of operation is incompatible with the serial processing model used by standard recursive ray tracers, which generate the ray tree depth first and hence require each ray to be traced through the scene geometry before another shader operation can be initiated.

Several possible approaches to solving this problem were looked at. To start with, the possibility of offloading some of the shading operations onto the GPU was considered. This would require the graphics hardware to take responsibility for deciding how child rays are spawned according to the BDRF of the material and the lighting setup in the scene. Unfortunately, this approach would require extensive and complex reprogramming of the GPU and would ultimately greatly reduce the feature set of our renderer. A second possible solution would involve pre-spawning rays for every possible interaction with scene geometry. Illumination would then be determined based upon their subsequent interaction with the volume. Although this method would leave the calculation of the reflection model with the CPU, many rays would be needlessly traced resulting in unnecessary load on the graphics hardware.

The next few sections discuss how the problem was approached.

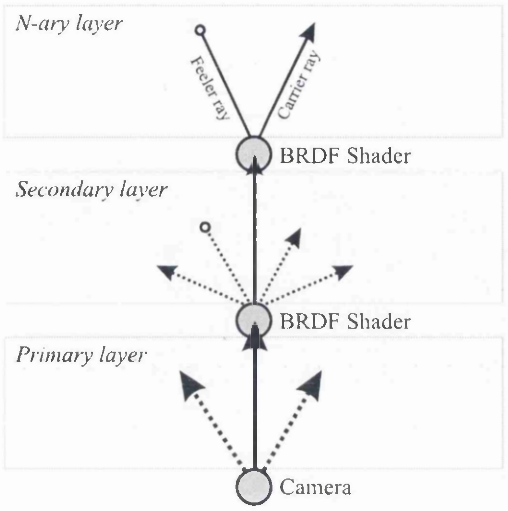


Figure 4.7: The Ray Tree

CPU Side

A two-pass, breadth-first construction of the ray tree (see figure 4.7) was decided upon, which leaves the computation of the lighting model on the CPU but also allows arbitrarily large bundles of rays to be sent to the GPU. The approach uses a ray cache which is utilized by the reflection and shading models used by the renderer. This approach allows the renderer to take advantage of hardware acceleration whilst still retaining maximum flexibility when incorporating new complex lighting models.

The algorithm begins by generating packets of primary rays at the camera and sending them to both the CPU geometry and GPU volume ray tracers. By using multi-threading, the renderer ensures that these two processes are carried out concurrently wherever possible. The nearest point of intersection along each ray is calculated and the reflection model associated with each hit point is executed resulting in a new layer of rays being spawned. The rays in this new layer are then traced and the process is repeated until either the maximum layer depth is reached, or the algorithm naturally terminates.

The second pass traverses the ray tree depth first in the conventional manner; only this time, no rays are traced and data is drawn from the cache. In addition to storing intersection data in each layer, the information about the state of non-deterministic processes such as the pseudorandom number generator is also stored. Restoring this information at each shading operation ensures that both passes will produce identical sets of results, even when Monte Carlo integration and Russian roulette sampling are used.

The pseudo-code for the algorithm is given in figure 4.6.

CPU to GPU Interface

The ray cache from the CPU ray tracer comprises of two blocks of memory, one containing the ray position vectors and the other containing normalised direction vectors (see figure 4.8). These bundles of ray data are each directly encoded as a 2D texture through OpenGL. To allow for arbitrary unclamped floating point data to be recovered from the textures on the GPU, `gl_texture_rectangle` is used the texture target and `gl_float_rgb32_nv` as the internal format. Similarly, to allow for unclamped floating point data to be written to the framebuffer, the framebuffer object OpenGL extension is used with the same internal texture target. The texture size is set to 1024x1024, giving a maximum of 1024^2 rays per bundle. The volume dataset is loaded into texture memory as a 3D texture once only at the same time as the first ray cache is loaded into texture memory.

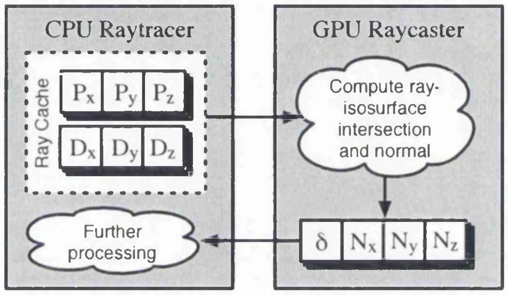


Figure 4.8: System Overview

A quad is placed in front of the viewport to act as a data stream generator [Har05]. Next, a fragment program on the GPU picks out the ray (P_v, D_v) pair from the encoded texture and marches through the volume in the direction of the ray until the specified volume isosurface is intersected. Any specified volume deformations are applied to the ray sample point before the volume is sampled, using Spatial Transfer Functions [CSW$^+$03]. If the isosurface is intersected for this ray, the normal at the intersection point is calculated using central differences. The final result is a quad of floats (δ, N_x, N_y, N_z) where δ is the depth of the intersection point along the ray, and N is the computed normal. This is written to the unclamped framebuffer as an RGBA result. If the ray does not intersect an isosurface within the volume, all values are set to 0 to inform the CPU raytracer of no hit.

The number of rays per bundle is kept to a multiple of 1024 to avoid the necessity of branching in the fragment program. This allows for rays to be ignored on a per-row basis by limiting both the size of the quad and the texture coordinates used. Once all fragments have been processed, all valid ray results are read back from the framebuffer into CPU memory. These results are then used by the CPU raytracer for recasting / shading.

Shader processing

The returned rays are processed by the CPU ray tracer, and for each intersection point, feeler rays to the light sources are placed into the ray cache as are any additional rays required by the shader functions (e.g. according to the BRDF or BSSRDF). This represents a new depth in the ray tree (figure 4.7), and once sufficient numbers are cached, these are sent for processing by the GPU.

4.5.3 Performance

This section reviews some of the results from the system, with comparisons given between our new CPU/GPU hybrid renderer (labelled *HYB* in graph) verses the entirely CPU-based renderer. The first test is a globally illuminated scene similar to that of figure 4.12 but without the box geometry (i.e. with the volume data set, light source, and base plane for soft shadows), and such that the head occupies about 80% of the scene. Results for this scene given in figure 4.9. This scene is designed to demonstrate the increase in speed of using a GPU renderer compared to just a software renderer. The GPU render times include the times to load the ray caches into 2D texture memory, and the one off time to load the volume into 3D texture memory.

The first test was designed to test the speed of intersections with the volume object by moving the camera close to the CT head dataset – such that over 80% of primary rays hit an isosurface. Table 4.9 shows the results for this scene.

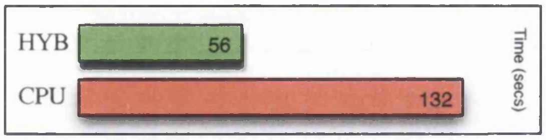


Figure 4.9: Render times for Scene 1

To measure the increased speed of load balancing with the GPU when the scene comprises of mixed geometry, the CT skull was placed inside a Cornell box (Figure 4.12). We first give timings for the box minus the volume object, to benchmark the standard CPU raytracer. A volume object is then placed into the scene, and benchmarked with CPU volume intersections and then GPU volume intersections. The *volume time* given in Figure 4.10 is the resulting difference between the base scene and the scene with the volume included.

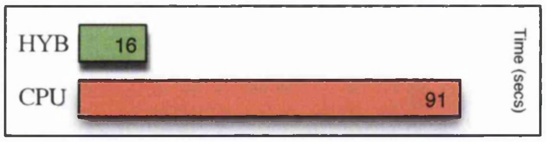


Figure 4.10: Render times for Scene 2

Although the GPU assisted rendering offers a good speed-up, it did not provide the expected speed-up. This is due to the fact that on the hardware setup used (a Geforce 6800) the upload of data to the GPU is extremely fast in comparison to readback performance. Therefore, although the ray cache can be uploaded to the GPU quickly, the results after processing by the GPU take a long time comparitively to read back. The performance of reading back data from the GPU is improving due to the increased demand of GPGPU applications and HDR techniques, so we anticipate that as the drivers mature, graphics cards will be able to offer vastly increased readback speeds, which will be comparable to upload speeds.

Another source of overhead in GPU based computation is that of branching operations. As reduced penalties for branching operations in the fragment shader and support for more optimal looping techniques are implemented, this will further increase the benefits of load balancing with the GPU.

A further consideration is the fact that the GPU's specialisation is parallel computation rather than flow control [Har05]. CPU's in comparison are highly optimised for flow control, with dedicated branch prediction units and other hardware optimisations. The result of this observation is that our GPU load balancing using ray caches performs to the strength of each unit. The GPU traces a large number of rays through the volume data, whereas the CPU makes conditional decisions about visibiliy feeler rays, material shaders, photon caching, etc.

Another advantage of computing the ray-isosurface intersections entirely in hardware is that complex deformation methods can be employed that would otherwise result in a massive performance penalty on the CPU. Figures 4.14 and 4.15 show the result of applying some spatial deformations to the volume objects. Notice the reflection of the deformed object in the teapot in each image. There was virtually no performance penalty in applying these deformations in hardware.

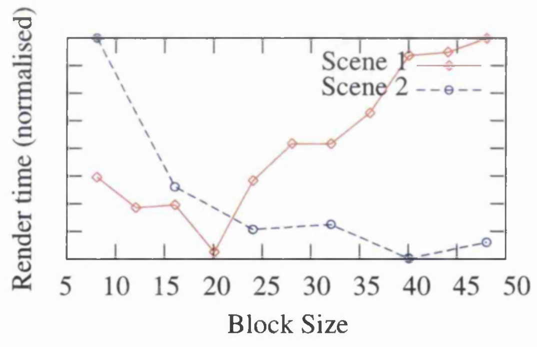


Figure 4.11: Render times with differing block size

Figure 4.11 gives the times to render two scenes with our system using different block sizes (render times have been normalised for clarity). *Block size* here is a variable parameter to the renderer that governs the number of primary rays per bundle in the first layer. It is clear from the results that the optimal choice of block size varies from scene to scene. Scene 2 in this example was a much more complex scene comprising of many more surface-based objects, and with higher numbers of hemisphere rays.

4.5.4 Method Conclusion and Future Work

This section has presented images demonstrating global illumination applied to volume data sets. Such techniques are extremely costly to compute for surface based graphics within ray tracing, with scenes taking hours to compute (for example rendering complex models such as the Sponza Atrium with path tracing or even photon mapping and final gathering), and with the size of volume data sets, this results in an even higher demand.

We acknowledge that for most visualization tasks, such GI renderings of volume data are not required, but just as GI renderings of surface based graphics are occasionally used for specific applications (e.g. architectural walk throughs), then GI renderings of volume data will be valuable in future. There are already many publications in the area, with methods such as those of Beason [BGB$^+$06] and Wyman [WPSH06], which can take days (on a single processor) to compute a 3D texture which is then used to produce real-time GI renders of isosurfaces. Such renders give more visual cues which are important for understanding the data, or could be used for publication; e.g. Beason has some excellent videos of a rotating functional MRI data set.

One of the features of the approach introduced in this section is that the renderer is taking advantage of GPU acceleration to speed-up the process of GI rendering of volume data. Therefore the approach will be of use for speeding-up precomputation of the 3D textures in the above methods. Other features of our approach are that surface geometry and volume data can be mixed, with the load split between the GPU and CPU. It has been shown that complex surface materials for volume data sets are possible, and additionally the system has integrated Spatial Transfer Functions [CSW$^+$03] to allow for complex deformations (such as the twisting (figure 4.14) and splitting (figure 4.15) shown) to be achieved efficiently and independently of the CPU.

There are still some improvements to be made to the method. A possible area of further research is that of estimating the optimal block size for a given scene; this optimal block size can be naively estimated before the main rendering stage by performing render tasks of a sparse set of blocks placed randomly in the final image, and measuring the results with differing sizes. More advanced heuristic approaches to this problem would likely gain larger benefits, it is anticipated that such approaches will give a a further increase in efficiency.

The multi-threading approach introduced in this section is an additional further area of research; particularly with the advent of multi-core CPUs and GPUs. Although it is often the case that currently both the CPU and GPU code are actively processing rays, there is no guarantee that both will complete their bundle at roughly the same time, resulting in a variable delays. Future work in this area will improve the multithreading model to ensure both CPU and GPU are closer to full capacity at all times. Also, by integrating the software and hardware volume renderers, it is possible to pass ray caches to either CPU or GPU algorithms for isosurface intersection. Therefore one extension to the method will be to allow unloaded cores (either GPU or CPU) to take bundles of rays from the cache and calculate them. This assumes that the volume object takes up the main computation. If geometry is the bottleneck, then systems such as the ray engine [CHH02] would allow the GPU to take on some of the geometry load.

Finally, figure 4.17 shows the bunny almost entirely illuminated by the focused caustic from the head, and figure 4.18 shows an augmented reality scene with GI and image-based lighting. These demonstrate the many advanced rendering techniques available for surface based graphics that the method introduced here has applied to volume based graphics.

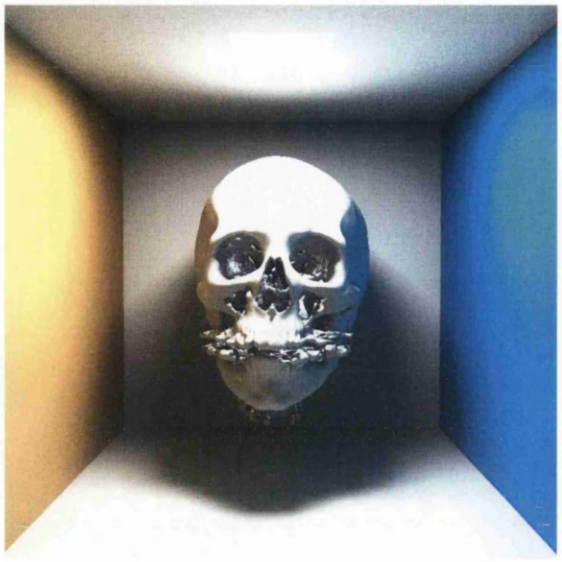


Figure 4.12: CT Head in Cornell Box

4.6 Summary

This chapter has been split into two main components : the *review* component and the *research* component, and is intended as a transition from the review chapters of the thesis to the research chapters.

The chapter began with a review of graphics hardware, the OpenGL pipeline, and an introduction to programmable graphics hardware; including details on the low-level aspects of providing the GPU with data and obtaining processed data after passes. Section 4.2 introduced the reader to the concept of using the GPU for volume rendering applications, which included a review of current methods but also an example implementation of the shaders

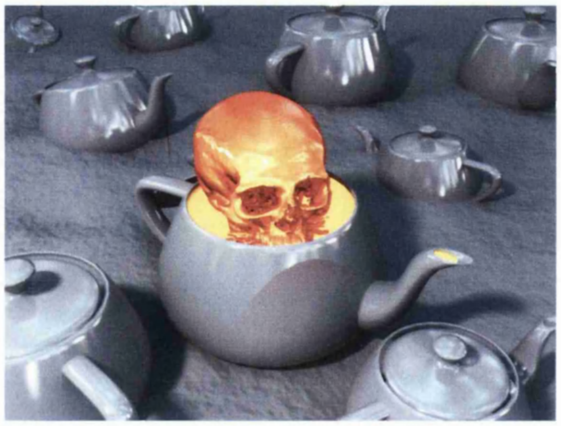


Figure 4.13: Skull in Teapot

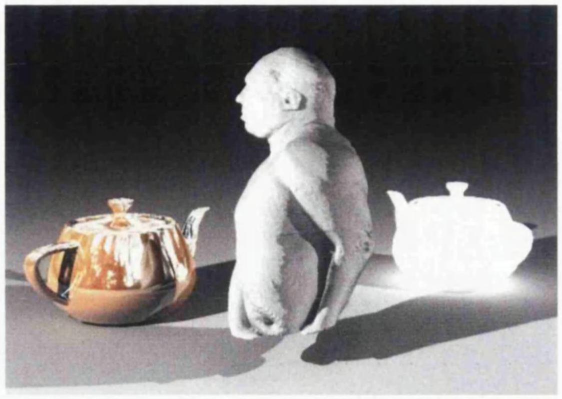


Figure 4.14: Volume deformation (twisting) on the GPU within GI scene

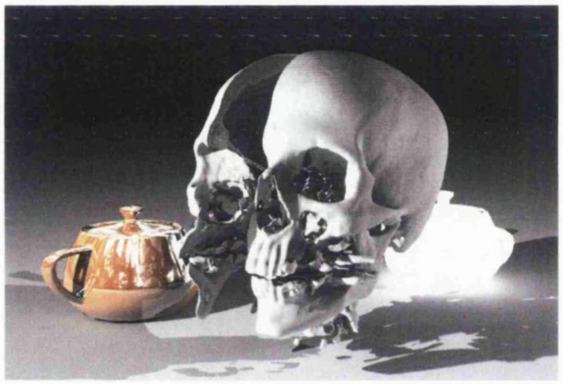


Figure 4.15: Volume deformation (splitting) on the GPU within GI scene

used with a rasterisation-based ray setup renderer. Point-based rendering was introduced in Section 4.4, including its history, applications, and details on how such rendering algorithms can be efficiently implementation on the GPU.

A new hybrid CPU/GPU load-balancing volume rendering application was introduced in Section 4.5, demonstrating the effectiveness of leveraging the power of the GPU for volume rendering applications. The method extends an existing ray tracing software application (Igneus [Spe]) to provide the GPU with bundles of rays to intersect with an isosurface inside the volume dataset. While there is much more potential research in this area, it is clear that the benefits of leveraging the power of the GPU can be great; particularly when the work can be load-balanced between the GPU and the much more flexible CPU and its associated resources.

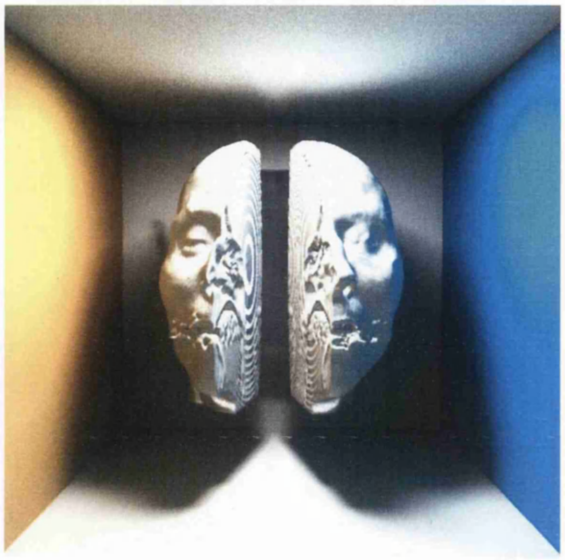


Figure 4.16: Volume deformation (splitting) on the GPU within Cornell box

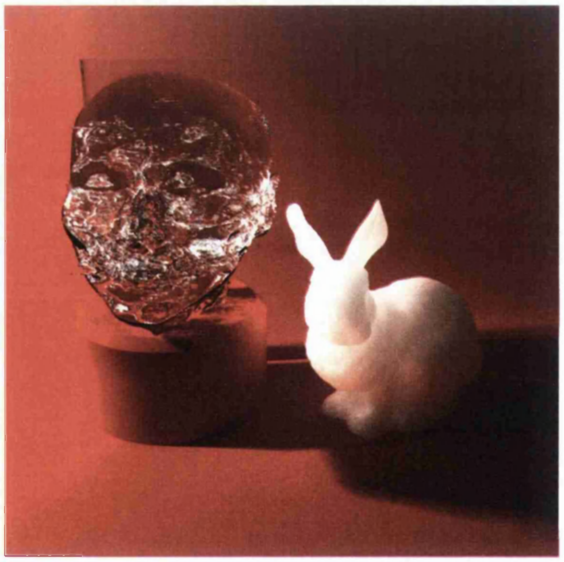


Figure 4.17: Translucent Bunny with Glass Head; rays: $\approx 486,000,000$

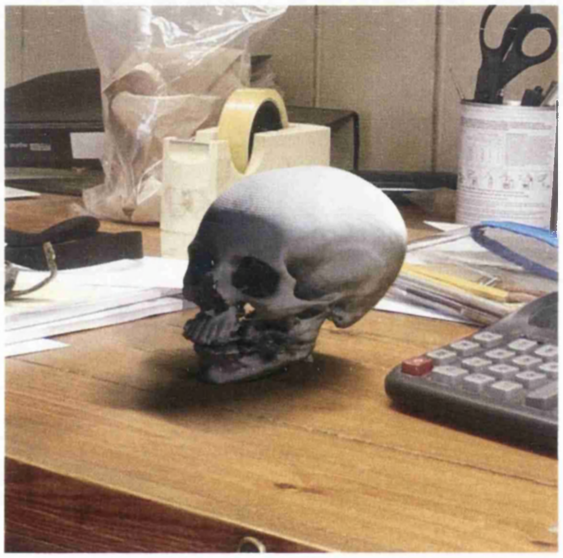


Figure 4.18: Skull on Desk; rays: $\approx 739,000,000$

Chapter 5

Volume Wires

Contents

Preceding chapters in this thesis have so far given an overview of volume graphics, including volume rendering techniques (Chapter 2), and methods for manipulating and deforming volume data (Chapter 3). The previous chapter gave an overview of GPU-based volume rendering techniques, and example implementations.

This chapter introduces the Volume Wires methodology for nonlinear volume deformation. The methodology is based upon the concept of using curve-skeletons (called *wires*) to deform the surrounding space of the volume object.

An initial study of this framework was presented at the Winter School of Computer Graphics (WSCG) 2006, and was additionally one of the 17 out of 62 conference papers accepted to the journal [WJ06], outlining the general concepts of the methodology and a CPU-based raycasting implementation of the rendering algorithm.

5.1 Introduction

Much work has been focused on methods of deforming volume datasets (see Chapter 3), but the majority of such research leans towards emphasising the low-level details of such algorithms rather than offering an intuitive interface to the user. By *intuitive*, we imply a method that does not concern the user with translating individual voxels and predicting the propagated effects of other voxels connected to it, at least when considering global deformation and character-based deformation. The low-level approaches such as those presented by Gibson [Gib97] (3D Chainmail) are suitable when wishing to treat the volume dataset as a soft, mouldable body, but not for when the volume dataset is to be treated as a globally deformable object. An intuitive methodology must also present a familiar object interaction metaphor to the user.

Much research has been focused on developing deformation systems for surface-based models for both static and animation purposes. Software tools such as Character Studio (a 3D Studio Max Plugin) [3ds] and Maya (owned by Autodesk) [May] allow the user to define skeletal systems for characters. These concepts have been carried over into the volume graphics domain with work done by Silver *et al.* [GS00b, GS00a, GS01] by producing a group of algorithms for the semiautomatic extraction of curve skeletons of the volume models and subsequent reconstruction. A hardware-based renderer also exists for rendering linear transformations of the skeletal segments [SSC03].

The Volume Wires framework presents an intuitive volumetric deformation methodology to the user, the *Volume Wires* methodology, and also details a group of algorithms for rendering such deformations and dealing with the mapping between object and world space.

5.1.1 Goals and Objectives

It is beneficial to maintain a standard raycasting pipeline for rendering the user-specified deformation, as the deformation rendering stage can be effectively integrated into an existing high-quality raycasting pipeline. The discretised volume rendering integral can be computed in world space by performing a backward-mapping operation (denoted as Φ^{-1}) on each sample point p_i along the ray. Many existing volume deformation techniques remove the advantage of compositing internal texture information by either rendering only a given isosurface or converting to a surface representation in an attempt to gain efficiency.

The full advantages of utilising an existing volume rendering pipeline are listed below:

- A specialised rendering algorithm is not required – many volume deformation systems require a rendering algorithm to be developed alongside the deformation algorithm. This dependence forces the developers to concentrate a vast amount of effort on creating hacks for correct perspective transformations, normal correction, and general image quality. As a result, the methodology is often hard to implement and integrate into another rendering pipeline.
- All the advantages of raycasting volume rendering are maintained, i.e. the availability of internal texture information. In addition, well-established ray compositing methods such as MIP are also possible.
- Any advanced effects built into the pipeline can be included also as the object is treated as being in world space. Accurate light transport [ARC05], and even global illumination (see Section 4.5).
- The feasibility of a GPU implementation is greatly increased, since many GPU-based raycasting volume rendering algorithms already exist, as detailed in Section 4.2 of

this thesis.

In addition, we wish to work with volume datasets natively rather than convert to some other representation (such as a triangle mesh, converted using the Marching Cubes algorithm) to preserve all of the discussed benefits of volume rendering. Although it is true that deforming a limited surface representation of a volume dataset would be more efficient due to the need of deforming only the vertices of the surface, such a representation would not provide the detail and image quality of the full volume dataset.

The aim of the Volume Wires framework is to evaluate the deformed volume object using backward-mapping in world space; the framework therefore is *non-reconstructive* in that no volume dataset is created from the deformation. The issues surrounding the reconstruction of deformed datasets have been discussed in Section 3.4.4.

5.2 Related Work on Swept Solids and Volumes

An extensive study of volume deformation techniques has already been completed in Chapter 3. Various solid modeling schemes were introduced additionally in Chapter 2. One area that was not discussed in detail however was that of sweep representations; it is necessary to give an overview of such representations as the work in this chapter can be considered as such. This section explores a combination of techniques relating to swept solids and swept volumes.

In surface graphics, algorithms have been devised to directly evaluate parametric space, such as that from Catmull [Cat75] which evaluates bi-cubic parametric surfaces by recursive subdivision of the surface into patches; the algorithm was unfortunately computationally expensive. Whitted [Whi79] describes an algorithm that evaluates parametric surfaces by employing first a bounding volume that encapsulates the entire object, and then recursively subdividing this into subvolumes. The ray-object intersection is then carried out recursively on the bounding volume first, and then the subvolumes. Kajiya [Kaj82] gives a more complex solution using algebraic geometry to transform the problem of ray-patch intersection into curve-curve intersection. Wijk [vW84] introduces a set of algebraic methods for calculating the intersection of rays with objects defined by sweeping planar cubic spline contours.

Sweep representations [BLWJ97, AMYO98, AMBJ02] are a solid representation used in CAD/CAM applications, existing alongside such representations such as Constructive Solid Geometry (CSG) and Spatial Partitioning schemes. A *swept object* is defined as the region occupied when a template object is swept through a trajectory through space [FvDFH96]. *Extrusion* is an example, and is a common operation in CAD software involving a linear sweep of some 2D template from one point to another along the axis perpendicular to the template's plane. A *rotational sweep* sweeps the template object around a given axis instead of an arbitrary trajectory.

More complex sweeps involving arbitrary trajectories (such as parameterised curves through Euclidian space) are termed *generalised sweeps*. The template object can be three-dimensional, and can be varied dynamically along the sweep. Interesting effects can be achieved by rotating and scaling the template depending on its position along the trajectory [Sea97], the former producing a twisting effect, the latter producing a warping effect.

One method of rendering swept volumes is to voxelise the result and render this new volume dataset using a standard direct volume renderer. The initial work on converting parametric curves, surfaces and volumes was completed by Kaufman [Kau87], though his method produces only binary volumes, leading to severe aliasing artefacts. This technique is implemented in by Chen and Winter [WC02], where 2D images are used as sweep templates. The voxelisation is achieved by recursively subdividing the trajectory, as in [Sea97]. The authors also give an approach to directly evaluating the resulting swept volume using numerical root finding techniques. This direct evaluation is however computationally expensive, and suffers from singularity problems, e.g. where an object is swept around an axis, deciding which points should be sampled on the axis itself is difficult.

5.3 Deforming with Volume Wires

Every volume deformation methodology must present some sort of interface to the user; for example, providing a means of 'grabbing' a voxel and pulling it and watching the effects of pulling this voxel on the other voxels, or perhaps by allowing them to use sculpting tools to warp the dataset. This section introduces the *wire* as a tool for deforming volume datasets.

5.3.1 Wire Definition

The central concept in the Volume Wires methodology presented in this chapter is the *wire*. A wire is any parametrically-defined 1D trajectory through 3D Euclidean space, e.g. Bézier curves, Catmull-Rom splines, or even simply a line defined by two points. The wire essentially acts as a curve-skeleton for the volume object that it is defined inside of – deforming the wire (by modifying the positions of the control points of that wire) deforms the surrounding volume object.

A wire is actually composed of two elements: the *object wire* and the *world wire*. The *object wire* defines the curve-skeleton of the object in object space; once defined, it remains static throughout the deformation process until the user decides that their initial skeleton definition requires modification. Associated with each object wire is the *world wire*. It is the role of the world wire to define the deformation of the target object defined by the object wire; therefore, there is a mapping defined between the space defined by the object wire and the space defined by the world wire.

To illustrate the wires concept, Figure 5.1 shows two example deformations using the Volume Wires methodology; on the left is a deformation defined on the Visible Human, and on the right is a deformation defined on the CT Carp. In the case of the Visible Human deformation, the *object wire* (shown as a red curve) defined by the user in this instance is acting as the 'backbone skeleton' or *spine* of the Visible Human, as the user has defined it roughly parallel with the spine. The associated *world wire* (shown as a green curve), has been deformed to produce a smooth bend; in this case the user has defined a bend of the

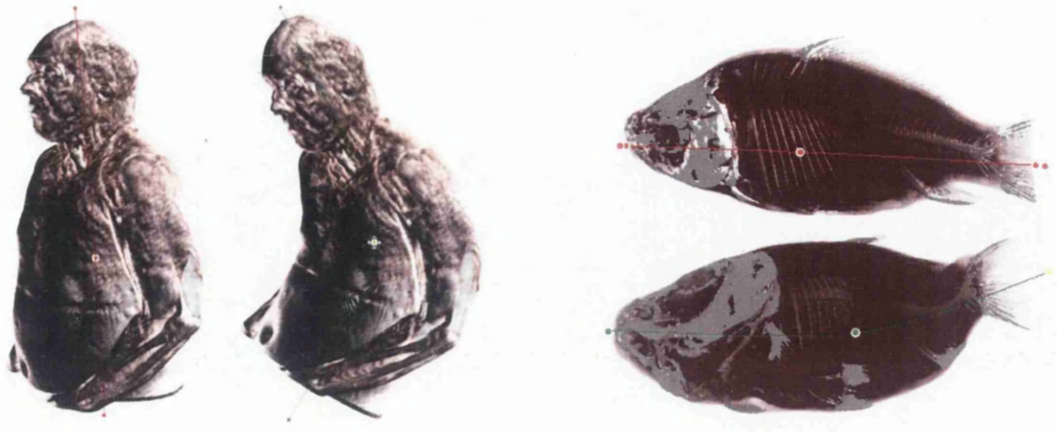


Figure 5.1: Two example deformations with the Volume Wires methodology on the Visible Human dataset and CT Carp dataset. The red curve is the object wire, and the green curve is the world wire.

Visible Human's spine. This image also shows the final render of the deformation, which has been rendered using a backward-mapping raycaster in world space.

In terms of how the volume object is deformed, the methodology can be considered in one of two perspectives:

- The wire deforms / warps the surrounding space of the wire, and consequently the volume object; or,
- The volume object is swept along the trajectory of the wire; specifically, the sweep is a type of *general sweep* along an arbitrary trajectory whose generating area changes through the sweep [FvDFH96].

For the majority of images in this chapter, Catmull-Rom cardinal splines are chosen as the wire trajectory definition. Our choice is based on the ability of Catmull-Rom splines to smoothly interpolate a given set of points, fitting nicely with the object-skeleton metaphor. In addition, Catmull-Rom splines are simple to work with, requiring no trajectory specifications. Catmull-Rom splines however do require two 'redundant' control points at either end as the spline is evaluated between only the 2^{nd} and $(n-1)^{th}$ segments (where n is the number of segments defined by two control points).

5.3.2 Wire Manipulation

A series of different effects can be achieved with the Volume Wires framework. The most simple, intuitive and immediately apparent effects are those of bending an object, as demonstrated in previous Figures. However, the wires can be used for purposes other than simple bends. Figure 5.2 shows the Lobster dataset. The topmost image shows the lobster with two object wires, each defined inside a claw. The middle image shows one world wire being moved away from the other, which has the effect of moving the associated claw away from the other claw. In this respect, the user is treating the system as a skeletal deformation system.

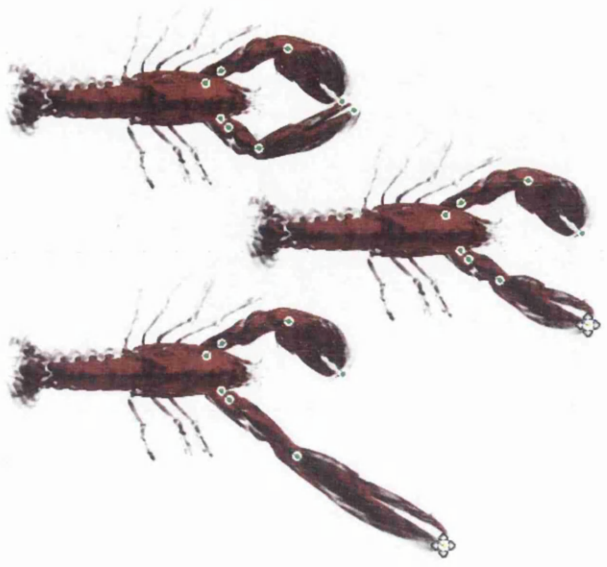


Figure 5.2: Lobster Manipulation

The advantage of the volume wires methodology for this application is that any cracks in the image (caused by skeletons segments linked via a common joint being deformed separately) are dealt with implicitly by the system. This is because the wires are defined in this case as Catmull-Rom splines, and thus have a curved boundary that warps the surrounding space in a smooth manner with no harsh transitions. In the case of deforming this lobster, the user has associated the voxels representing each claw to a particular wire; these voxels remain 'attached' to the wire as the wire is deformed. It should be noted however that this functionality requires segmentation information to correctly label each claw; such segmentation functionality can be integrated neatly into the Volume Wires framework, and will be discussed in Chapter 6.

The bottom image in Figure 5.2 shows a claw being stretched, achieved by simply pulling one control point of the world wire towards the right. Because the rendering algorithm used in this example image is a backward-projection renderer, holes in the image caused by voxels being stretched apart are not apparent, as the values in-between samples are approximated using trilinear interpolation. Note that in the case of stretching the data, if the rendering algorithm were a forward-projection algorithm such as Splatting [Wes90], then cracks would be apparent in the final image where voxels had been pulled apart by the world wire. Any rendering algorithm implementing a forward-projection view of the deformation would need to account for this issue.

Another simple operation with a wire is to translate the entire wire around in the scene, which has the effect of translating the deformed volume object to this new position. Alongside these mentioned implicit effects, the Volume Wires framework additionally defines two explicit effects that can be applied to world wires. These effects are *warping* and *twisting*,

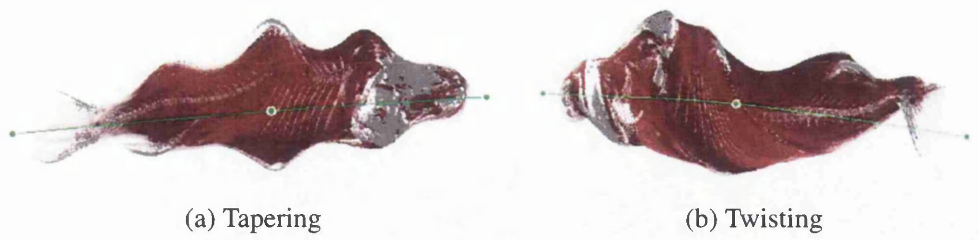

(a) Tapering (b) Twisting

Figure 5.3: Tapering and twisting effects applied to a wire controlling the CT Carp dataset

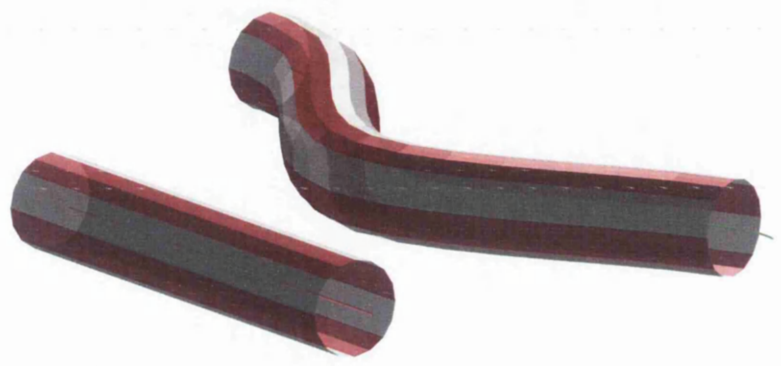


Figure 5.4: Mapping a striped polygonal tube from the object wire (left) to the world wire (right)

and correspond to offset-dependent scaling and rotation transformations respectively; warping is a scaling that varies along the length of the wire, and twisting is rotation that varies along the length of the wire.

Figure 5.3 shows the CT Carp dataset with *(a)* a tapering effect, and *(b)* a twisting effect.. with rotation and scaling values associated with it to achieve twisting and warping effects. We refer to these values as effect values. The effect values can be added at the control points of the wire, and are linearly interpolated along the length of the wire.

5.4 Deformation Evaluation

The Volume Wires framework defines a non-reconstructive deformation evaluation system, whereby the final image is generated using the deformation data and the target volume dataset rather than a volume dataset that has been reconstructed (revoxelised, in this context) in the deformed state. This section introduces the overall objectives of the deformation evaluation stages of the framework and therefore the *mapping* between object space (where the dataset is defined) and world space (where the overall scene is defined).

Figure 5.4 gives a polygonal analogy of the Volume Wires deformation framework, whereby a tube is deformed by placing an object wire (red, on the left) inside and defining the associated world wire (green, right) to have a distinct curve in its shape. The result is that the tube

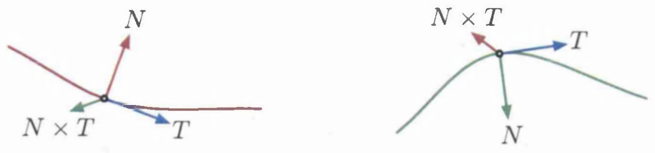


Figure 5.5: Correspondence between Frenet frames on the object wire and the world wire.

has been deformed 'around' the world wire.

Since the deformed volume in the Volume Wires methodology can be considered a swept volume, each point in both spaces is associated with its nearest wire point. A discussion on mapping techniques is given in Section 3.4.2. The below gives the series of steps required to compute a forward and backward mapping of the deformations in the Volume Wires framework.

- A *forward-mapping* function $\Phi : \mathbb{E}^3 \longrightarrow \mathbb{E}^3$ must discover, for each voxel P in object space, where P should be relocated in world space; therefore it is a mapping from the object wire to the world wire.
- A *backward-mapping* function $\Phi^{-1} : \mathbb{E}^3 \longrightarrow \mathbb{E}^3$ must discover for each point P in world space where the the sample point in object space should lie; therefore it is a mapping from the world wire to the object wire.

An inherent symmetry exists between the forward and backward mapping operations in the framework. Since the goal of the mapping process is to map points from one wire to another, the mapping can be reversed by switching the roles of the wires:

- A forward mapping operation maps voxel samples $v \in \mathbb{E}^3$ from object space to world space; and therefore from the object wire to the world wire;
- A backward-mapping operation maps samples $p \in \mathbb{E}^3$ from world space to object space; and therefore from the world wire to the object wire.

To keep the text coherent, we discuss only the forward mapping operation here. We use the following *notation* throughout the text:

- $w_o(t)$ and $w_w(t)$ give the computed points on the object and world wires at parametric offset t, respectively;
- $w_{rot}(t)$ gives the rotation effect angle at offset t;
- $w_{scl}(t)$ gives the scaling effect magnitude at offset t.

As discussed, the backward-mapping of the discussed operations below can be achieved by switching the roles of the wires.

5.4.1 Mapping Objective

Since the mapping is achieved by mapping from one wire's Frenet frame to the other, a procedure is required to locate a suitable wire offset given a point in space. The most suitable correspondence between samples $p \in \mathbb{E}^3$ and wire offset t is to find the object wire offset t which minimises the distance between p and $w_o(t)$. This can be achieved naively by scanning each wire offset and storing the offset that minimised the distance. A further discussion on the problem of computing the minimally distant offset for general curves can be found later in Section 5.6, which additionally gives a method of encoding such information into a volume dataset.

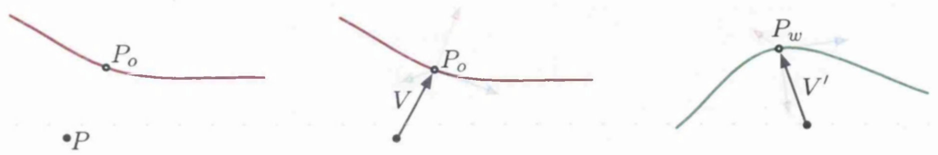


Figure 5.6: Mapping P from the object to the world wire. Note that $P_o = w_o(t)$ and $P_w = w_w(t)$ for the respective t-values found to minimise the distance.

Algorithmically, the steps required to map a point P from object space to world space (and consequently, from the object wire to the world wire), are given below, and are illustrated in Figure 5.6

1. Discover P_o, the nearest point on the object wire to P (at parametric offset t)
2. Compute vector $V = P - P_o$
3. Compute (using t) corresponding point on world wire P_w
4. Transform V from P_o's Frenet system to P_w's Frenet system

The next section discusses the local coordinate systems defined for the wires so that the correspondence can be formalised.

5.4.2 Frenet Frame Correspondence

The text above gives a series of steps that must be performed to map a point in object space to world space based on the object and world wires. In order to compute this mapping, an association must be made between the coordinate systems defined on each wire. The mapping between wires is achieved by mapping points / samples from one wire offset's defined Frenet frame [Blo90] to the other. Frenet frames are defined as the orthonormal basis of three moving vectors T, N, B defined along the trajectory of a space curve:

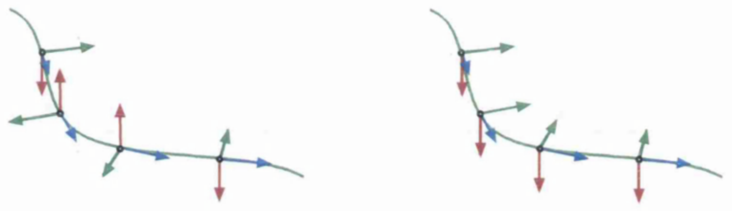
- $T(t)$ is the tangent of the curve at offset t, and is is defined as the first-order derivative of the curve;
- $N(t)$ is the principal normal vector, and is the second-order derivative;
- $B(t)$ is orthogonal to $T(t)$ and $N(t)$ and is therefore $T(t) \times N(t)$.

Figure 5.5 shows (left) a Frenet frame defined on a Catmull-Rom spline, with $T(t)$ shown in blue, $N(t)$ shown in green, and $B(t)$ shown in red.

In the discrete case where a wire is approximated as a set of points, an approximation can be obtained at point index i as:

$$\begin{aligned} T_i &= \frac{w_{i+1} - w_i}{\mid w_{i+1} - w_i \mid} & (5.1) \\ N_i &= \frac{T_{i-1} \times T_i}{\mid T_{i-1} \times T_i \mid} & (5.2) \\ B_i &= N_i \times T_i & (5.3) \end{aligned}$$

The main problem with using the second derivative for computing N on arbitrary offsets is that the normals are not 'stable' enough to be used for intuitive modeling – N can flip suddenly at inflection points (where the sign of the curve changes), producing a sudden, harsh transition. This problem is discussed by Winter in a PhD thesis [Win02] and Bloomenthal in two research papers [Blo85, Blo90]. It is illustrated in Figure 5.7.



Arbitrary normal calculation Precalculated and corrected normals

Figure 5.7: A comparison between (left) calculation of Catmull-Rom normals using the second derivative at arbitrary points, and (right) precalculated and corrected normals by rotating successive Frenet frames to match trajectory vectors.

The solution to this problem of unstable normals is adopted from work by Bloomenthal [Blo90], whereby the Frenet frames are calculated successively along the curve and corrected to avoid flipping; the method is also adopted by Winter *et al.* [WC02, Win02] for correcting normals in Bézier trajectory sweeps.

To calculate corrected normals, an initial Frenet frame for offset $t = 0$ is first computed. This Frenet frame is then placed at each successive wire offset t_{n+1}. At each new offset t_{n+1}, the frame is rotated such that the frame's trajectory vector T_{n-1} matches the trajectory vector at t_{n+1}, T_{n+1}. The rotation axis is given as $T_{n-1} \times T_n$, and the rotation angle as $\cos^{-1}(T_{n-1} \cdot T_n)$. These corrected normal vectors can be stored in a lookup table for later retrieval; to compute a corrected Frenet frame at any arbitrary point on the wire, an interpolated wire normal can be retrieved from the table with a chosen interpolation kernel. For a GPU implementation, the corrected normals can be encoded into a 1D texture for input to the shaders.

As discussed in Section 5.3.2, rotation and scaling values can be applied to points on the world wire, which give a twisting and tapering effect respectively. Figure 5.8 shows a functional twisting effect and a tapering effect defined and rendered on a surface-based tube using the MATLAB tool suite. The specification of these effects has to be taken into account during the mapping of one coordinate system to another.

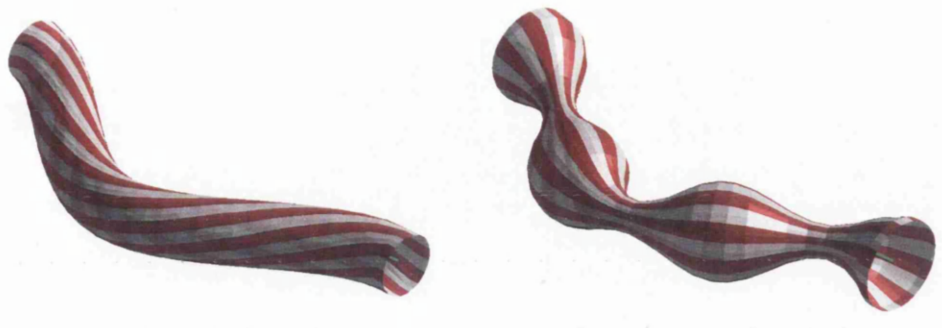

(a) Twist as $\sin(t * 20) + 2$ (b) Scale as $cos(t * 4) + 2$

Figure 5.8: Twisting (rotation) and Tapering (scaling) effects defined on the world wire

To achieve greater efficiency during the mapping process, the observation can be made that roughly the same transformations will take place on vectors that are matched to the same wire offset. Therefore, a preprocessing step can rotate the normal vectors defined in the normal lookup table at each offset t by $w_{rot}(t)$ around the axis defined by the trajectory vector. This avoids the necessity of computing the rotation of vector V each time as the coordinate system itself has been pre-rotated. The scaling effect defined along the wire trajectory is relatively cheap to compute, involving only basic multiplication of the coordinates, and would not benefit from such optimisation.

The next two sections give two possible mapping computation approaches for the Volume Wires methodology. The choice of mapping approach is very much dependent on the where the bulk of computation is performed and the performance of the computations on different sets of hardware. This is particularly true when considering CPU verses GPU implementations, where different programming styles must be adopted and consideration given to dedicated hardware implementations of important mathematical functions.

Orthogonal Matrices

The first mapping approach is to use the properties of orthogonal matrices to first transform the object wire's Frenet frame system to the common origin axes, and then back to the corresponding Frenet frame on the world wire.

Two matrices M_o and M_w are defined which transform the origin axes into the object wire and world wire Frenet frames, respectively:

$$M_o = \begin{bmatrix} & & & 0 \\ \mid B_o \mid & \mid N_o \mid & \mid T_o \mid & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_w = \begin{bmatrix} & & & 0 \\ |\,B_w\,| & |\,N_w\,| & |\,T_w\,| & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where T_o,T_w are the object and world trajectory vectors, N_o,N_w are the precomputed object and world normal vectors from the precomputed lookup table, and $B_o = N_o \times T_o$, $B_w = N_w \times T_w$. We also assume the existence of $S(p \in \mathbb{E}^3)$ which is a homogeneous scaling matrix with scaling factor p, and $T(p \in \mathbb{E}^3)$ which is a homogeneous translation matrix with translation amount p.

Given the vector V, the closest point P_o on the object wire, and the corresponding point P_w on the world wire, the forward-mapping operation is:

$$p' = T(P_w) \cdot M_w \cdot S(w_{scl}) \cdot M_o^t \cdot V$$

where a superscript t denotes the matrix transpose. The backward-mapping operation is given as:

$$p' = T(P_o) \cdot M_o \cdot S(\frac{1}{w_{scl}}) \cdot M_w^t \cdot V$$

These matrices can be pre-multiplied algebraically into one matrix if desired, although we found that this gave a substantially larger number of computations in this case due to redundant computations in each matrix cell.

Frame Rotation

The second approach to mapping the vector V to its new coordinate system is slightly more expensive to compute on the CPU, since it requires the calculation of trigonomic ratios. However, this assumption is based on the speed of acquiring such ratios being relatively slow; modern GPUs such as the GeForce 6800 onwards can retrieve both the sin and cos of an angle in one clock cycle.

A universal rotation matrix [Piq90] that can rotate a point around an arbitrary rotation axis a is given by:

$$R_{a,\theta} = \begin{bmatrix} ta_x^2 + c & ta_xa_y + sa_z & ta_xa_z - sa_y & 0 \\ ta_xa_y - sa_z & ta_y^2 + c & ta_ya_z + sa_x & 0 \\ ta_xa_z + sa_y & ta_ya_z - sa_x & ta_z^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where for brevity in the matrix, $s = \sin(\theta)$, $c = \cos(\theta)$, and $t = 1 - \cos(\theta)$.

Given the object and world trajectory vectors T_o and T_w, object and world normal vectors N_o and N_w, the rotation axis is derived as:

$$a = T_o \times T_w$$

with the rotation angle for forward mapping:

$$\alpha = \cos^{-1}(T_o \cdot T_w)$$

and backward-mapping simply with α negated.

The rotation effect is already taken into account from the precomputed normal vectors in the normal lookup table. The scaling amount s is applied finally after the rotation to give the final derivation:

$$p' = S(w_{scl}(t)) \cdot R_{a,\alpha} \cdot V$$

5.4.3 Correct Normal Calculation

Since a deformation is applied to the scene and the resulting dataset is not reconstructed but defined implicitly, each normal used in lighting calculations must be transformed beforehand from object space to world space according to the deformation.

The normal can be brought into world space by performing a subset of the mapping calculations. First, the normal N is obtained from the dataset via central differences. The new normal N' in world space is then given as:

$$N' = M_w \cdot M_o^t \cdot N$$

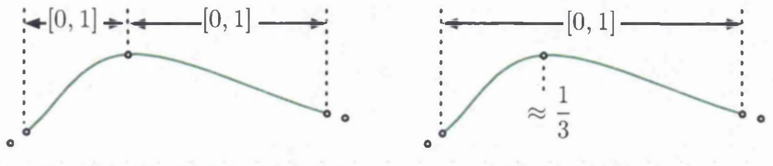
that is, the standard mapping calculation, but with the translations removed. This gives an effective and simple method for transforming the normals from object to world space without resorting to Jacobian matrices. This method was can be computed more efficiently than setting the fourth component of the normal to zero to ignore the translations.

For a backward-mapping scheme, there exists one more method for computing the new normal; the world space being sampled can be treated like a reconstructed volume dataset by performing Φ^{-1} on each sample point. Using this knowledge, the central differences operation to infer the normal can be performed in world space: applying Φ^{-1} to six points in world space, sampling at each backward mapped point, and using these new points in the central differences equation. This scheme is reasonably accurate (within discrete limits) but suffers from the computational expense of six extra computations of Φ^{-1} on top of the computation for the central point. A discussion on normal calculation in this manner has been given in Section 3.4.3.

5.4.4 Wire Memory Addressing

For Catmull-Rom splines (which are used throughout this chapter in result images), the individual segments defined between the second and last but one control points are evaluated individually in $[0, 1]$. To fit Catmull-Rom splines neatly into the mapping operations discussed, the t-value $[0, 1]$ should be 'spread' through the spline. This could be achieved by dividing the spline into fixed precision segments. However, this scheme is clearly non-optimal where the segment length varies greatly as each segment would be given the same precision; segments with greater lengths would suffer from a lack of precision compared to their smaller neighbours.

For the Catmull-Rom splines used in this and following chapters, the $[0, 1]$ range is spread over the wire evenly based on the length of each segment. This implementation decision has particular benefits if the wires are to be discretised into a fixed series of points, as the distance between discrete points will be more consistent and therefore maximises the precision available.



(a) Default Catmull Evaluation (b) Desired Evaluation

Figure 5.9: Catmull-Rom spline evaluation

Figure 5.9(a) shows the default evaluation for Catmull-Rom splines, where individual segments (defined by two interior control points) are evaluated in $[0, 1]$. 5.9(b) shows the implemented evaluation scheme for the Volume Wires system, where the $[0, 1]$ range is spread through the spine. Note that the control point in the middle now lies roughly at $t = \frac{2}{3}$.

This evaluation method can be implemented neatly in C++ by defining an abstract base class of type *Curve* and then defining a child class *CatmullRomCurve* which implements the methods for evaluating the curve. A member function is defined that takes an offset t and returns the correctly-evaluated point along the curve. Methods can be additionally added for defining any attributes along the trajectory of the wire (such as the scaling and rotation values for the tapering and twisting effects) by simply defining a new lookup table for each attribute and offering an interpolated value for given t-values along the curve.

5.5 Calculating Deformation Boundary

The deformation specified by the user will have a bounding box that is defined by both the deformation itself and the size of the volume dataset. In the case of forward-mapping the deformation, the boundary can be calculated simply by keeping track of the minimum and maximum coordinates of each point after applying function Φ. However, as discussed in Section 3.4.2, there is no trivial process for computing the deformation boundary on-the-fly using a backward-mapping technique, as it is not certain whether each sample point on the ray will map to the volume dataset until Φ^{-1} is actually applied.

A common operation in volume graphics is the separation of background voxels from the rest of the data. In CT/MRI datasets, the air surrounding the patient would produce very low scalar values because the air is less dense than the solid material that the patient is composed of. It is useful therefore to define a threshold value β which defines the border between background and non-background voxels in the target dataset. All voxels greater than or equal to this threshold are said to be β-voxels.

In the next section, a method of encoding the deformation into a volume dataset, called the *mapping field*, will be discussed. Before this encoding is possible however, a method must be devised for computing a bounding box of the specified deformation to ensure that there is minimal waste; it would be trivial to provide a 'good enough' bounding box, but a better solution would be to ensure a tight-fitting bounding box that can be computed without too much computational overhead.

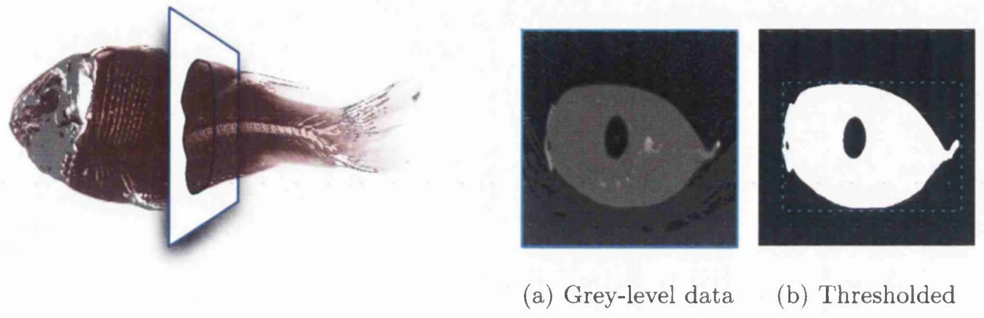


Figure 5.10: A slice taken along the carp with (a) The slice image, (b) A threshold of 0.25

5.5.1 Forward-mapping slices

An approximate bounding box of the current deformation specified by the user in the Volume Wires methodology can be obtained by forward mapping slices of the data from object space to world space. Figure 5.10(left) shows a slice defined along the object wire of the CT Carp dataset; the resulting slice image is shown to the right with *(a)* the slice image retrieved directly from the scalar data, and *(b)* the slice image with a thresholding operation to delineate background from non-background voxels. A threshold value of around $\beta = 0.25$ was decided upon based on trial and error, as it was seen to remove the packing material existing around the fish at the time it was scanned; though it also removed the hollow region inside the fish's stomach.

This slicing approach could be used to estimate the deformation boundary based on β. A set of planar slices can be defined along the object wire, aligned with the tangent of the wire. The slices are sized automatically in-place to tightly fit the data defined by β, as shown by the blue border in Figure 5.10(b).

If these slices are now forward-mapped onto the world wire, then an approximate bounding box is estimated by computing the bounding box of the new slices. The slices can be forward-mapped by applying Φ to each of the slice's geometry vertices.

5.5.2 Incorporating Slice Masks

There is an amount of redundancy caused by this approach in that the slices in this implementation have a set rectangular shape; in theory however, arbitrary shapes are possible. One simple approach to reduce the redundancy is to incorporate *slice masks* into the system. The slice mask associated with each slice is defined as a 2D bitmap of the thresholding result – the bit is set to 1 if the voxel is above or equal to β, and 0 otherwise. If the snapshot is obtained from the slice on the object wire and mapped to its new position on the world wire, a rough binary approximation to the deformed object is obtained.

The slice masks for the deformation need only be computed when a change is made to the object wires in the scene. The resolution of the slice masks can be varied; however for simplicity, the mask resolution is set to the size of the associated plane in our implementation. The computation of the slice masks using nearest-neighbour interpolation is extremely efficient and adds no significant computational time to the bounding box calculation stage.

To further optimise the bounding box estimate, this slice mask can be traversed with pointer arithmetic while 'attached' to the forward-mapped slice in world space. The final bounding box defined for the deformation is now based the set of all forward-mapped slice mask texels with value 1.

Alternatively, the slice mask information can be used to provide a segmentation facility.

5.6 The Mapping Field Encoding of the Deformation

This section introduces a set of algorithms for 'encoding' the deformation data used in the mapping process into a discrete 3D dataset, which is referred to in coming text as the *mapping field.*

According to the mapping operation given in Section 5.4, the datum required for each sample point p_i on the ray at render time is the offset of the closest point on a wire in the scene. If a forward-mapping operation were required for a forward-projection renderer, then the desired offset would be to the closest object wire point. Conversely, if a backward-mapping operation were required for a backward-projection renderer, the required offset would be the closest world wire point.

The coming text describes the mapping field generation process in a projection-independent manner for simplicity. An encoding using the object wires would facilitate a forward-mapping, since each voxel in object space can be associated with its nearest wire point and subsequently translated to its new position in world space; an encoding using the world wires would facilitate a backward-mapping since each sample point p_i in world space can be associated with its nearest world wire point and backward-mapped into the dataset (in object space).

5.6.1 Locating the Closest Point on a Wire

The main reason that encoding the deformation into a mapping field is desirable is that discovering the closest point on the wire can be an expensive operation; the closest point on a 1D line segment can be discovered easily using the equation of the line. However, discovering the closest point on a parametrically-defined curve is a nontrivial problem if the curve is cubic.

There is unfortunately no general, closed-form analytic solution to finding the closest point on cubic curves such as Bézier curves and Catmull-Rom splines. Schneider gives a numeri-

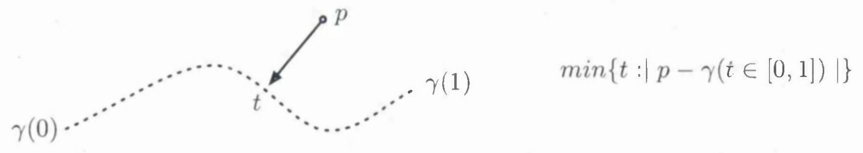


Figure 5.11: Closest point on curve

cal root-finding algorithm for finding the closest point on a Bézier curve [Sch90a, Sch90b]. The algorithm exploits known properties of Bézier curves (such as the curve always lying within the convex hull of its control points) to subdivide the curve and find the closest point. This algorithm is implemented by Chen and Winter [WC02] to evaluate sweeps of 2D templates along Bézier trajectories at render-time. This algorithm, however, is still expensive as it requires numerical root finding methods and thus becomes impractical for render-time evaluation at interactive frame rates. Winter gives an additional commentary on the subject of finding the closest point to a cubic spline in his PhD thesis [Win02].

A brute-force approach to computing the closest point on the curve might involve recursively subdividing the curve into smaller and smaller segments until a given tolerance is reached, but such approaches are unpredictable both in the time they take to compute the closest point and in the accuracy of the final result, particularly for curves with sharp bends.

5.6.2 Mapping Field Representation

Conceptually, encoding the deformation into a discrete 3D dataset is similar to 'dependent texture' techniques used in volume deformation on graphics hardware, such as that used by Rezk-Salama *et al.* [RSSSG01], where the deformation is encoded by means of displaced texture coordinates. The key difference between dependent textures and the mapping field is that the mapping field stores the parameters to a mapping function instead of displaced texture coordinates.

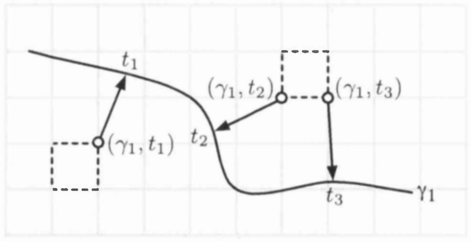


Figure 5.12: A 2D slice through the mapping field

At each voxel in the mapping field is a tuple (γ, t) where γ gives the identifier for this voxel's nearest wire, and t gives the nearest offset along the wire given by γ. These values are used at render-time to evaluate the deformation, with full details of this process (the *mapping process*) given later in this chapter. [1]

[1]For efficiency of memory usage, just one floating-point value would suffice by defining a $[0, 1]$ space as the

5.6.3 Mapping Field Creation

The mapping field is created using a distance propagation technique (see Section 2.5 for a discussion on distance propagation techniques). As discussed, an observation of distance propagation techniques in general is that alongside the minimum distances, any attributes of the surface (e.g. colour) can additionally be propagated from their original locations, outwards towards the dataset boundaries, such that the distance field becomes:

$$D(p) = (min\{\mid p - q \mid : q \in S\}, a_1, \ldots, a_n) \tag{5.4}$$

where $a_1, \ldots, a_n$ are the additional attributes at point p. Looking up a value in such a field gives not only the minimal distance to an object of interest, but also the attributes that were found at this closest point.

In a traditional distance field, the distance value at each voxel represents the minimal distance to some isosurface τ. For a Volume Wires deformation, it is not an isosurface that is of interest; instead, the minimally-distant wire offset is the value that is required at each voxel. Therefore, voxelised versions of the wires themselves are to be propagated instead of a chosen isosurface value. Specifically, the values to be propagated are the (γ, t) values denoting the wire reference and the wire offset, respectively. The field now becomes:

$$A(p) = (min\{\mid p - q \mid : q \in W\}, \gamma(q), t(q)) \tag{5.5}$$

The distance values are discarded once the propagation algorithm has completed, saving 4 bytes per voxel for floating-point accurate fields; the distance values are not required for any of the rendering algorithms discussed in coming sections.

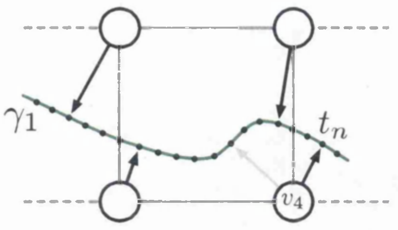
5.6.4 Propagating the (γ, t) Attributes in the Mapping Field

Before distance propagation techniques can propagate the distances (and in this case, the additional attributes (γ, t)), an initial set of voxels must be initialised at points of interest; in this case, the points of interest are a finite series of points lying on each wire.

Initially, the distance values of the voxels in the mapping field are set to ∞. As with general distance propagation, an initial set of voxels must now be initialised with default (γ, t) values in order that these values can be propagated outwards towards the edges of the mapping field.

Algorithm 2 gives the full method for pre-propagation initialisation, and Figure 5.13(a) gives a 2D illustration of the process. The objective of the initialisation phase is to ensure that each voxel belonging to a cube that the wire traverses is aware of its nearest point on the wire. In order to achieve this, the wire is incrementally followed through the field at a specified arbitrary precision of n points $wp_0, wp_1, \ldots, wp_{n-1}$.

space of all discrete wire offsets; or a 32-bit integer could be used to store both values using bit masking and bit shifting.



(a) Voxel Cube Calculation

$$\begin{aligned} \gamma &= \gamma_n \\ t &= t_n \\ d &= \mid v_4 - \gamma_n(t_n) \mid \end{aligned}$$

(b) v_4 data

Figure 5.13: Voxel Cube Pre-Propagation Initialisation

Once a set of points inside a voxel cube is established, the algorithm computes for each of the cube's voxels, the closest wire point to that voxel. If this minimal distance value is less than the current value stored at the voxel, then the distance value is updated along with the (γ, t) pair. This is illustrated in Figure 5.13(a) as an arrow pointing from a voxel to a point on the wire. Figure 5.13(b) shows a snapshot of the data that is maintained for voxel v_4.

Note that it is possible that a voxel may be updated with a wire point inside a different cube than was previously stored. This is shown in Figure 5.13(a), where voxel v_4 has just been updated with its new nearest wire point t_n. The previous choice of closest wire point is shown as a greyed-out arrow, and the new current data for the voxel is shown in Figure 5.13(b).

Once this initialisation process is complete, the initialised distances and (γ, t) values are propagated using a distance propagation algorithm. Our preference is to use a vector distance transform instead of a standard distance transform, due to its increased accuracy in this application (a visual comparison between distance and vector techniques is given later in this chapter). Mullikin's 3D vector distance transform algorithm [Mul92] is used for the propagation, which propagates vectors pointing to the closest wire point, resulting in increased accuracy at the expense of slightly increased memory usage due to the requirement of storing vector data instead of just distance data. This increased memory requirement however was found not to be a problem with today's machines; and additionally the algorithm only requires the storage of three slices' worth of vectors at any one time.

5.6.5 Analysis of Mapping Field Methods

In this section, a CPU-based raycasting volume rendering algorithm utilising backward-mapping is used to compute the Volume Wires mapping operations and produce a rendered image of the deformation, complete with lighting and a 1D transfer function.

The focus in this section however is not on the implementation of the renderer; a complete GPU-based rendering algorithm is given to complete the chapter in Section 5.7. The focus here is give a brief analysis of how varying mapping field variables (namely, its size and the choice of distance field creation technique) affect the final image quality.

Mapping Field Size

The size of the mapping field has a large effect on the quality of the rendered image, due to the increased number of voxels available for trilinear interpolation; a higher number of voxels gives a more accurate result, but at the expense of greater memory usage and increased computation time.

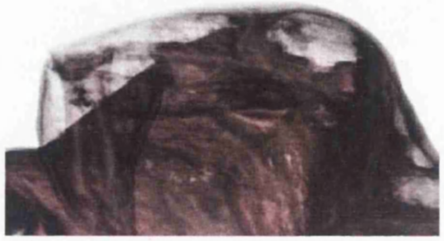
Figure 5.14 shows a comparison between images rendered with differing sizes of mapping fields, all computed using the Chamfer distance transform using a $3 \times 3 \times 3$ quasi-Euclidean matrix. At the top is an image rendered with a full size mapping field – that is, a $1 : 1$ correspondence in scale between the volume dataset and the mapping field. This image is used as a reference image for the three comparisons below, which show final renders with mapping fields of reduced size; i.e. the first row shows a render achieved from a mapping field with 1% of the voxels of the reference mapping field, taking 0.01 seconds to compute. Note how the image difference between this render and the reference render is greater than that of the increased size mapping fields below.

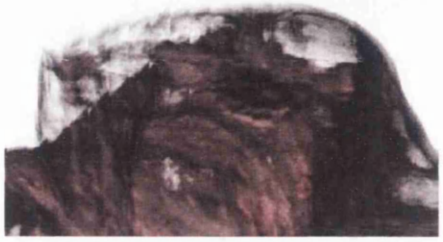
Distance verses Vector Transform

The naive method of computing the mapping field would be to compute, for each voxel, the nearest point on any wire in the scene; essentially this would be an Euclidean distance field. This process is extremely expensive, but creates an optimal result.

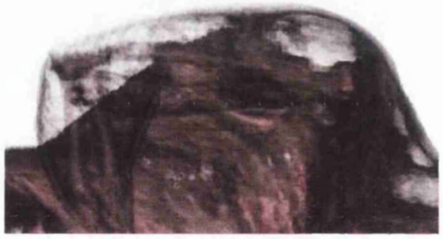
Figure 5.15 shows a comparison between the naive approach and two different propagation methods, namely the Chamfer transform with a $3 \times 3 \times 3$ quasi-Euclidean matrix, and Mullikin's EVDT algorithm. Each comparison is computed with three sizes of field, with the field size (and therefore accuracy) increasing down towards the bottom of the page. For each rendered image computed from a propagated field, we give the image difference (negated for clarity) and root mean square between it and the image rendered using the naive field.

As expected, the vector transform significantly outperforms the chamfer transform; it is clear from the images that the EVDT algorithm gives the highest quality results of the two propagation methods, based on measured error and visual difference from the naive method.

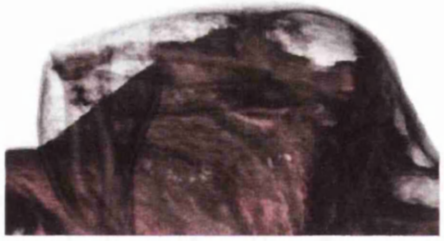

Reference image (6.05s)


(a) 1% (0.01s)

Reference difference


(b) 3.7% (0.17s)

Reference difference


(c) 12.5% (0.76s)

Reference difference

Figure 5.14: Deformations encoded into varying mapping field to volume dataset scales. Scale numbers given in percentage of full-scale mapping field. The time to propagate the field is given alongside. Difference images have been negated for clarity.

Algorithm 2 Pre-propagation Wire Freezing

```
Function      Freeze
Description   Main entry-point function for the algorithm
Parameters
    arrWirePts   An array of wire points
1: for all points $p$ in $arrWirePts$ do
2:    if a new cube is entered then
3:       $FreezeNeighbours(wireID, arrPlots)$
4:       $arrPlots.clear()$
5:    end if
6:    $arrPlots[plotCount].(point, offset) \leftarrow (arrWirePts[i], i)$
7: end for
```

```
Function      FreezeNeighbours
Description   Takes an array of plots within one cube (bounded by eight voxels) and sets
              all neighbours to reference their minimally-distant point.
Parameters
    wireID     Current Wire ID
    arrPlots   Array of plots
1: $minDist \leftarrow \infty$
2: $minOffset \leftarrow 0$
3: for all neighbouring voxels $v$ in current cube do
4:    for $i = 0$ to $arrPlots.size()$ do
5:       $d \leftarrow \mid arrPlots[i].point - (x, y, z) \mid$
6:       if $d < minDist$ then
7:          $minOffset \leftarrow p$
8:          $minDist \leftarrow d$
9:       end if
10:   end for
11:   $mappingField[v].\delta \leftarrow minDist$
12:   $mappingField[v].t \leftarrow minOffset$
13:   $mappingField[v].\gamma \leftarrow wireID$
14: end for
```

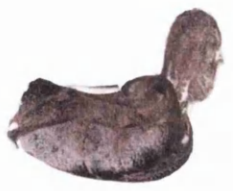
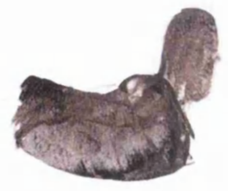
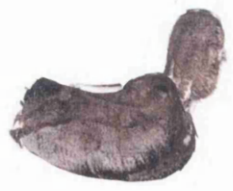
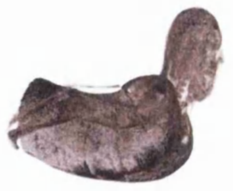
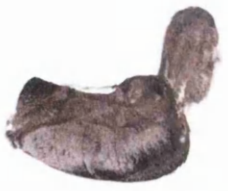
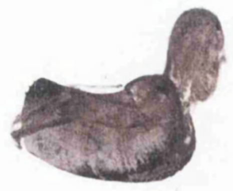
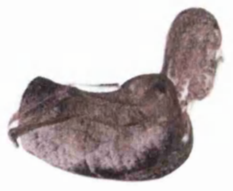
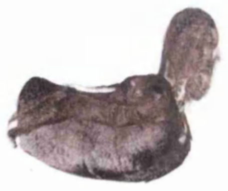
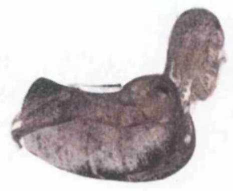
	Naive	**Chamfer**		**EVDT**	
Field Size		**Render**	**Diff. with Naive**	**Render**	**Diff. with Naive**
$16 \times 19 \times 25$					
	0.59 secs	< 0.001 secs, RMS 5416.52		< 0.001 secs, RMS 3216.53	
$32 \times 37 \times 55$					
	4.68 secs	0.05 secs, RMS 3923.87		0.08 secs, RMS 2475.83	
$62 \times 72 \times 109$					
	35.83 secs	0.4 secs, RMS 3468.00		0.6 secs, RMS 1736.62	

Figure 5.15: A comparison of the Chamfer ($3 \times 3 \times 3$ quasi-Euclidian matrix) and EVDT distance propagation techniques with the most accurate, but brute force naive approach. Given for each approach is the time taken to compute the field for the given field size, and for the propagation methods, the root mean square of the image difference.

5.7 A GPU Backward-Mapping Raycaster

Before highly-optimised rendering algorithms for the Volume Wires methodology can be discussed in coming chapters of this thesis, it is necessary to lay the groundwork for a reference GPU implementation of a raycasting volume renderer that uses backward-mapping on ray sample points p_i to compute new sample points p'_i in the volume dataset. These sample points at p'_i are then sampled, and the discovered values used in the transfer function and subsequently the compositing operation.

The implementation introduced in this section highlights the advantages of using a backwards-mapping raycaster; many of the deformation algorithms discussed in Chapter 3 employ specialised rendering algorithms and often do not offer fully customised compositing operations. The Volume Wires framework can be integrated into a standard raycasting pipeline more easily.

This section introduces a GPU-based raycasting volume renderer that evaluates the Volume Wires mapping field to discover new sample points in the volume dataset. The end result is a dynamically-computed image of the deformation running at interactive frame rates.

5.7.1 Mapping Field Raycasting & Backward-mapping

A standard CPU-based raycasting volume renderer can be modified quite simply to project its rays into the mapping field $\mathbf{M}$ instead of the volume dataset $\mathbf{V}$, and call the inverse mapping function $\Phi^{-1}(p, \gamma, t)$ for each sample point along the ray.

```
for all Rays $r$ do
    for all Sample points $p_i$ on $r$ do
        $p_{mf} = p_i \cdot M_{world \to field}$
        $(\gamma, t) = \mathbf{M}(p_{mf})$
        $p' = \Phi^{-1}(\gamma, t)$
        $sample = \mathbf{V}(p')$
        $composite(sample)$
    end for
end for
```

Before this can be achieved however, a mapping between world space and mapping field space must be established to allow the renderer to locate the correct point in the mapping field for each world space sample point. Figure 5.16 shows the required coordinate system mappings for the rendering process. Rays cast into world space (5.16(a)) are first mapped into mapping field space (5.16(b)). Since the mapping is affine, only the ray P_v and D_v vectors need to be mapped, and all sample points in mapping space can be linearly interpolated by simply adding the required difference when traversing the mapped ray. The matrix required to map points from world to mapping space is defined as:

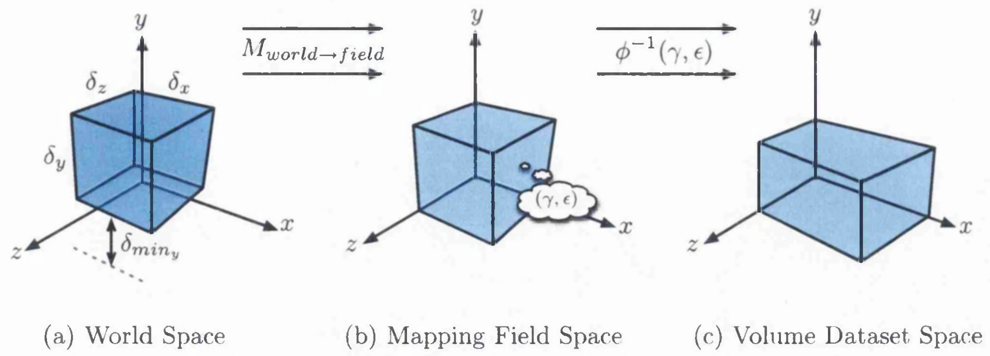


Figure 5.16: Mapping from world space to the volume dataset

$$M_{world\rightarrow field} = \begin{bmatrix} \frac{\lambda}{\delta_x} & 0 & 0 & \frac{\lambda}{\delta_x} - \delta_{min_x} \\ 0 & \frac{\lambda}{\delta_y} & 0 & \frac{\lambda}{\delta_y} - \delta_{min_y} \\ 0 & 0 & \frac{\lambda}{\delta_z} & \frac{\lambda}{\delta_z} - \delta_{min_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

where λ is the field size, and $\delta = \delta_{max} - \delta_{min}$.

For each sample point inside the mapping field, the (γ, t) pair is obtained. These values are now sent to the backwards-mapping function Φ^{-1} along the the current sample point p_i on the ray in world space. Φ^{-1} gives a new sample point p_i' which becomes the position in the volume dataset to sample (Figure 5.16(c)). The volume dataset is now sampled at this point, and the value is used in the compositing scheme that is being used for the current rendering.

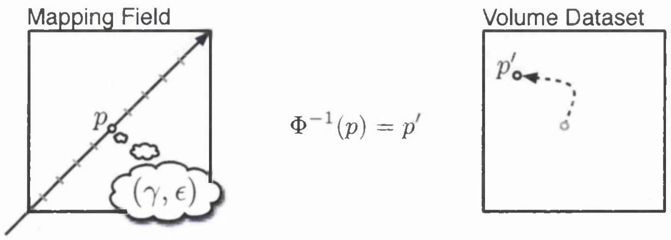


Figure 5.17: Sampling the mapping field at each ray sample point and performing Φ^{-1}

The advantage of using a backward-mapping function for the deformation rendering is clear – once the function is implemented, it is relatively straightforward to implement it into an existing volume rendering pipeline. In addition, the advantages of volume rendering such as internal texture information, shadows, etc are all maintained.

5.7.2 Challenges of GPU Backward-Projection with Deformation

Many GPU-based raycasting algorithms exist for volume rendering, and a review of the most popular algorithms can be found in Section 4.2. Early GPU-based volume rendering algorithms used a slice-based approach where slices (aligned either with the viewer or the dataset) were textured with the corresponding volume dataset slices; the volume rendering integral is approximated in this case by the hardware blending the slices together. Later generations of consumer GPUs brought programmable shaders; the ability to perform trilinearly interpolated 3D texture lookups and dynamic looping are of particular importance to volume rendering as traditional raycasting can be performed on such hardware.

The most optimal manner in which to generate rays is to use a dataset rasterisation technique, as introduced by Westermann *et al.* [KW03] and discussed in detail in Section 4.2.1. By rasterising the front and back faces of the volume dataset, the ray entry and exit points in world space can be inferred by interpreting the resulting framebuffer $< R, G, B >$ values as world coordinates. These values can be written to a texture as input to a fragment shader which then performs samples along the ray. At each sample point p_i along the ray, Φ^{-1} is applied to give p'_i, the new sample point in the volume dataset.

It is immediately apparent however that such a scheme would only make sense where a static volume dataset is rendered; if a real-time deformation is applied to the dataset, then points inside the volume dataset boundaries may be transferred by Φ 'outside' the boundaries. It is generally not possible to judge whether a given ray will 'contribute' to the final image - i.e. whether sample points $p_i, \ldots, p_{i+1}$ will backward-map (using Φ^{-1}) to sample points $p'_i, \ldots, p'_{i+1}$ within the volume dataset boundary. Rasterising the volume dataset to generate the ray entry / exit points therefore would be unreliable; some method for approximating the boundary of the deformation is required so that this boundary can be rasterised instead (such as the method given in Section 5.5).

A technical issue for consideration when performing a backward-mapping function on each ray sample point is the maximum instruction limit for fragment shaders. When this limit is hit, the GPU writes out the contents of its debugging registers to the framebuffer. The maximum primitive instruction limit on GPUs such as the Geforce 8800 is 2^{16}, which can be surpassed easily by performing high amounts of computation inside loops; if Φ^{-1} is compiled to around 512 primitive GPU instructions, then the number of ray steps needs to be less than 128 to avoid hitting this limit, and this is without taking any auxiliary computations into account such as the compositing operations, ray operations, etc. Fortunately, the high-level shading languages Cg and HLSL provide details on the instruction counts to assist the developer. The developer can however achieve many more instructions per pixel by performing multiple passes.

5.7.3 Algorithm Overview

The GPU raycasting algorithm introduced in this section uses a rasterisation-based ray setup stage. Since the rays defined in the world are to be fired into the mapping field, the boundary of the mapping field (mapped to world space) is rasterised to generate the ray entry and exit points for the fragment shader.

The raycasting and backward-mapping operations discussed previously in this chapter are computed in the fragment shader on the GPU. Because of the computational power of the GPU, the mapping operation is computed many times faster than the equivalent number of mapping operations computed on the CPU.

The following GPU functionality is used for the algorithm.

1. Framebuffer Objects (FBO) and attached depth RenderBuffers (with 16-bit floating-point accuracy) – provide a means of writing intermediate results such as ray entry/exit points and compositing information, to a texture;
2. Fragment discarding and Early Z-Culling – an important means of providing efficient early ray termination;
3. Multiple Render Targets (MRTs) - this functionality allows the graphics hardware to write colour values to more than one render target. In this case, the render targets will be separate Framebuffer Objects;
4. Runtime looping (not unrolled at compile time) / branching in fragment shader.

A discussion on these technologies in the context of GPU volume rendering is given in Chapter 4 of this thesis. During the coming discussion, we denote the three framebuffer objects used as FBO_0, FBO_1, and FBO_2. A particular framebuffer object may be used more than once for a different purpose throughout the rendering process, and is usually cleared to all-zero before use. The data used in the mapping operation is encoded as described in the next section.

5.7.4 Data Encoding

In order for the fragment shader to compute the deformation backward-mapping, it requires access to the volume dataset, mapping field, and the wires. The manner in which the data is encoded for the GPU therefore deserves discussion.

Volume Dataset & Mapping Field

The volume dataset and mapping field are both encoded as 3D textures. The volume dataset encoding is trivial, and simply involves setting the correct dimensions and data format in OpenGL, with the scalar value semantically mapped to the red component of the texture.

The encoding of the mapping field however requires special attention since it is actually comprised of two values per voxel: (γ, t). One possible encoding of the mapping field would be to use an $< R, G, B >$ internal format where γ is mapped to the red values and t to the green values. This however introduces some redundancy in the blue component, and additionally, consideration must be given to the fact that the volume dataset must also exist in GPU memory at the same time.

A more space-efficient GPU encoding of the mapping field can be realised with a single value floating-point field, with the full range of each wire's t-value spread through the $[0, 1]$ range. The t value can be recovered from the mapping field by multiplying the mapping field value by the number of wires and subtracting the floor of the the result (which in turn becomes γ).

Object and World Wires

Along with the nearest wire point obtained from the mapping field at each ray sample point, the fragment shader requires access to the wire information, including the precomputed normals (as per Section 5.4.2) for input into the mapping operation.

Consideration must first be given to whether the wires are discretised before encoding. The advantage of pre-discretising the wire is that the logic required to turn t-values into the final point on the wire is shifted to the CPU; GPUs are designed to perform large amounts of simple computations, but not flow-control and logic operations. Since the t-value will be used to compute a wire point, the GPU will be required to calculate a wire point given a t-value once for each active voxel in the scene. Therefore, a great deal of computation can be saved by precomputing the discrete wire representation into a relatively small number of points beforehand, as the request for a wire point is reduced to a memory address computation.

The discrete wire points can be provided to the GPU as an $n \times 2m$ 2D texture, where n is the number of discrete points per wire (referred to in future as the wire *precision* and m is the number of wires in the scene. Each wire occupies two rows in the texture: one row for the object wire points, and one row for the world wire points. Each $< x, y, z >$ wire point occupies one texel by encoding the coordinates into the $< R, G, B >$ values of the texel; the alpha component is utilised for the scalar value representing the scaling effect factor at that point.

The number of points the wire is discretised into (and subsequently the number of columns in the texture) is set to a power of two to provide the greatest lookup speed [2]. In addition, the precomputed and corrected wire normals are encoded into a separate 2D texture in the same manner.

Since the textures are used as data storage, the interpolation is set to nearest-neighbour to ensure that no interpolation of values occurs between the wires in each row of the texture. To improve the results of computing points along the wire in the shaders however, linear interpolation can be implemented cheaply by providing the shaders with a function for interpolating between wire points (columns) but not the wires (rows). The Cg code for achieving this is given below.

Listing 5.1: Linear interpolation for sampling a wire

```
float3 wireSample(
   uniform sampler2D wire,
   float2 pos)
{
   return lerp(tex2D(wire, pos),                        // sample a
      tex2D(wire,float2(pos.x+(1/WIRE_PREC), pos.y)),     // sample b
      frac(pos.x*WIRE_PREC));                          // [0,1] offset
}
```

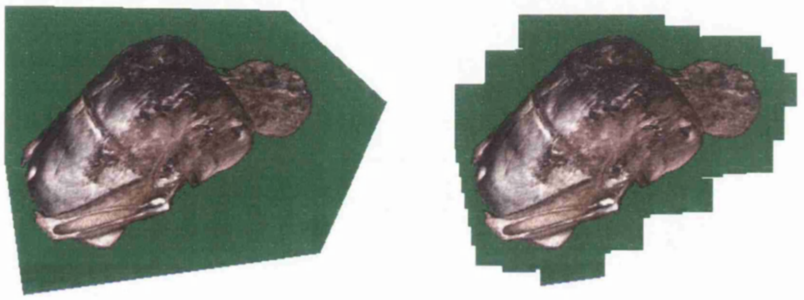
where $WIRE\_PREC$ is the precision of the wire.

[2]The GPU is highly optimised for fetching data from power-of-two sized textures – though modern GPUs have the ability to fetch from non-power-of-two textures if required

5.7.5 Identifying 'Valid' Rays

The method given in Section 5.5 for determining the boundary of the deformation can additionally provide an important optimisation for the rendering stage, by providing an approximation of the image-space projection of the deformed object. The planes and associated plane masks used in the process can be used to mark voxels in the mapping field that contain 'valid' data; that is, data that backward-maps to voxels above the β threshold. Voxels in the mapping field that do not satisfy this property are referred to as $\varnothing$-voxels in coming text.

During rendering, many rays will potentially travel through the mapping field and hit only $\varnothing$-voxels. The optimisation task therefore becomes the identification of rays (or bundles of rays) that will hit one or more valid voxels in the mapping field before they exit; the elimination of such rays will help avoid redundant computation of Φ^{-1}. We denote such rays *valid* rays, and all others $\varnothing$-rays.



(a) Without VRI (b) With VRI

Figure 5.18: Usage of Valid Ray Identification. The green area shows rays that have hit only $\varnothing$-voxels.

This process is referred to as *Valid Ray Identification*, and its effect is shown in Figure 5.18. To eliminate as many $\varnothing$-rays as possible from being fired, the mapping field is first divided into large blocks (e.g. 20^3), and blocks containing one or more valid voxels are identified. For each of these 'valid' blocks, a point in space centred at each active block is added to a list of valid blocks. These points can now be rasterised at a sufficiently large size to give a rough estimation of the active areas of the final image. All framebuffer pixels containing valid rays are guaranteed to be within this pixel subset (along with a comparatively small number of $\varnothing$-rays around the perimeter). The manner in which the rays are eliminated from computations is using the Z-buffer, and the complete process is given in following sections.

5.7.6 Preparing to Raycast – Ray Setup

The first stage in the rendering algorithm is to generate the ray entry and exit points based on the mapping field's position in world space.

Z-culling is utilised for the elimination of unnecessary computation by eliminating fragment computation on fragments which are defined to belong to these invalid rays. A caveat of using Z-culling for this purpose is that in order to enable it, the state of depth computation cannot be modified after being cleared. This means that once the depth function is chosen, it cannot be altered to perform subsequent passes – all passes must work with the same depth function. In addition, the depth cannot be altered in the fragment shader. However, the Geforce 6600 used during development allowed fragments to be discarded without disabling Z-culling.

Stage 1 - Rendering the valid points

Depth testing is initially disabled. The points generated from the block list (as discussed in section 5.7.5) are rasterised to FBO_0. In order to use the depth buffer to cull the $\varnothing$-rays, the depth buffer must contain 0 for $\varnothing$-rays, and 1 otherwise. When this is the case, all fragments falling into a $\varnothing$-pixel (in turn corresponding to a $\varnothing$-ray) will be immediately eliminated without being processed.

The below table shows the state of the depth buffer and three Framebuffer Objects. A greyed-out entry indicates that the buffer state has not changed as a result of the previous operation.

Buffer	**Current contents of pixel** p
Depth	Initialised to a maximum depth of 1
FBO_0	$p_{red} = \begin{cases} 1 & \text{if } p \text{ was hit by a valid-block} \\ 0 & \text{otherwise} \end{cases}$
FBO_1	Initialised to 0
FBO_2	Initialised to 0

Depth testing is now switched on, and depth function is set in OpenGL as `GL_GEQUAL` meaning that incoming fragments will only be further processed if their depth is greater than or equal to the current depth buffer value. The depth is initialised to 0.

A viewport-sized quad is drawn (a 'data-stream generator' in GPGPU terminology), and a fragment program picks out the values from FBO_0 to determine how the depth is set. If there is a value of 1 for the red component, then this implies that this pixel was hit by a valid-block, so the fragment is discarded (using the `discard` keyword in Cg). Otherwise, the fragment is kept and the colour is set to an arbitrary value, since it is not used in subsequent stages. The below table shows the current buffer state.

Buffer	**Current contents of pixel** p
Depth	$\begin{cases} 0 & \text{if } p \text{ is a } \varnothing \text{ pixel} \\ 1 & \text{otherwise} \end{cases}$
FBO_0	Cleared to 0
FBO_1	Initialised to 0
FBO_2	Initialised to 0

Stage 2 - Ray Setup

This stage involves performing a rasterisation-based ray setup stage on the mapping field. The mapping field boundary coordinates are first brought into world space. The six faces of the boundary are each defined as a `GL_QUAD`.

The front faces are drawn by culling all back-facing faces. A fragment program running on the GPU writes the incoming texture coordinate to the framebuffer as the $< R, G, B >$ colour value, and these values are stored in the attached framebuffer object FBO_0. These texture coordinates represent the normalised world coordinates of the boundary of the mapping field.

Buffer	**Current contents of pixel p**
Depth buffer	$\begin{cases} 0 & \text{if } p \text{ is a } \varnothing \text{ pixel} \\ 1 & \text{otherwise} \end{cases}$
FBO_0	Texture coordinate (world position in [0,1])
FBO_1	0
FBO_2	0

Next, the back faces are drawn by culling all front-facing faces. This time, the fragment shader takes in FBO_0 as an input and uses it to compute the ray direction given the current texture coordinate. The ray direction is written to render target FBO_1's $< R, G, B >$ values, and the ray length is written to the α component. The ray length is computed as the distance between the two rasterised points, which correspond to the ray intersection points.

The table below gives the final buffer state for the ray setup stage. FBO_0 and FBO_1 will be used as input textures to the raycasting stage, providing the necessary ray parameters for each pixel.

Buffer	**Current contents of pixel p**
Depth buffer	$\begin{cases} 0 & \text{if } p \text{ is a } \varnothing \text{ pixel} \\ 1 & \text{otherwise} \end{cases}$
FBO_0	Texture coordinate / world position in [0,1]
FBO_1	p_{rgb} : Normalised ray direction, p_a : Normalised ray length
FBO_2	0

5.7.7 GPU Raycasting

The raycasting stage consists of multiple passes with an initially fixed number of ray steps per pass. FBO_0 and FBO_1 are used as input textures to the raycasting stage, providing the necessary ray parameters for each pixel. FBO_2 is used as a compositing buffer, storing the accumulated colours and ray opacities. Therefore, FBO_2 is used both as an input texture and as the (single) render target[3].

[3]The composition calculations benefit from the high precision and dynamic range offered by the internal format used by the Framebuffer Object (16-bit per component).

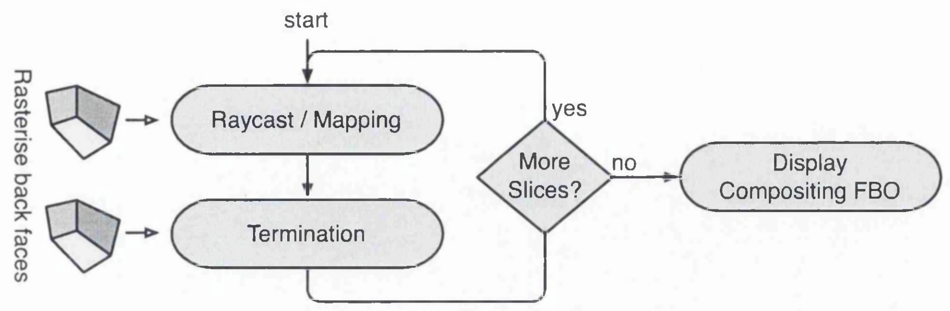


Figure 5.19: The iterative raycasting process

Figure 5.19 shows the iterative raycasting process. Each pass is invoked by rasterising the back faces of the bounding box. The resulting fragments will pass the depth test (set to greater than or equal) as they were the last rasterised primitives (from the ray setup stage).

A fixed number of these passes is made based on the ray step. Each pass is actually composed of two stages controlled by two different fragment programs – the *ray traversal shader* and the *termination shader*. The ray traversal shader traverses each ray, performing the backward mapping operation Φ^{-1} and accumulating the colours and opacities. The termination shader then checks if the ray should be terminated based on its opacity value – if the opacity is above 0.95, then the ray is terminated by setting the depth value to 1 to ensure the fragment belonging to the ray fails the depth test in future passes.

These two stages are split into two in order that the GPU can perform early-Z termination, which is a capability of the GPU to avoid invoking the fragment shader if it can guarantee that the fragment will fail the depth test; to qualify for this guarantee, the shader must not modify the fragment depth itself. For this reason, the GPU code that decides when a ray is to be terminated is kept in a separate, dedicated shader.

Performing Φ^{-1} in the Fragment Shader

At each ray sample point p_i in world space, the world space to mapping field space matrix (given in Equation 5.6) is applied to give the mapping field sample point. The mapping field is next sampled at this point, and the value obtained is used to derive the (γ, t) pair required for the wire point lookup.

The following information is obtained from the textures:

- P_o and P_w – the nearest object wire point and the associated world wire point;
- T_o and T_w – the object wire and world wire trajectories;
- N_o and N_w – the precomputed object wire and world wire normals.

The parametric offset t is already normalised and therefore addresses the correct texel. The (γ, t) values are first used to obtain the object and world wire points P_o and P_w. The normalised trajectory vectors for the object and world wires T_o and T_w are obtained from the discrete approximation of the first derivative by subtracting the computed wire point from the next point in the texture and normalising the result; achieved by adding a small amount $\epsilon = 1/wirePrecision$ to the offset to obtain the next wire point. Additionally, the wire normals N_o and N_w are obtained from the wire normals texture.

Once these six vectors have been obtained from the texture information, Φ^{-1} is computed as per the mapping derivations given in Sections 5.4 to obtain the final sample point p'_i in the volume dataset. The volume dataset is now sampled at p'_i and the value used a lookup function encoded as a 1D texture to obtain an $< R, G, B, \alpha >$ value to be included in the compositing equation.

Ray Termination

The number of ray steps is only modified near the end of the ray in the case where the number of steps would take the ray out of the volume. Listing 5.2 gives the Cg code for computing this.

```
// calculate the distance from ray origin to the current ray position
float currDist = distance(rayOrigin,rayCurr);
// calculate the distance from ray origin to where it would end up
// after MAX_STEP steps
float finalDist = distance(rayOrigin,rayCurr+(MAX_STEP*rayDir));
// if that takes the ray out of the boundary...
if(finalDist >= rayLength) {
   // be sure the opacity is set to 1 when done (termination condition)
   terminateAtEnd = true;
   // calculate the final number of steps to avoid redundant calculations
   stepCount = ((rayLength-currDist)/(finalDist-currDist))*MAX_STEP;
}
```

Listing 5.2: Cg code for calculation of ray length

5.7.8 Results

A selection of result images are now shown, along with the total time to render them (i.e. the sum of both mapping field generation and the time to generate the image on the GPU). Additionally, to judge the speed of the GPU algorithm, we give the approximate number of frames per second achieved when interacting with the resulting deformation using a trackerball-like interface to rotate the camera around the data with a 512×512 framebuffer.

Figure 5.20 shows four frames from an animation of the CT Knee dataset. The animation wires were specified for a total of 24 frames in less than a minute, and the time taken to generate the 24 mapping fields and produce the images was 10.43 seconds on a P4 3.4GHz with a GeForce 6800. Once the mapping field was created, navigation of the deformed data in world space was performed at around 4FPS.

Figure 5.21 shows three frames from an animation of the CT Carp dataset. The animation was specified not through user interaction, but by defining the movement of the control points as a sine wave, producing a travelling sine wave deformation in the carp's body. The total time for mapping field generation and rendering of the 24 frames was 12.8 seconds. The rendering speed was calculated during interaction with the deformed data as approximately 4FPS.

Figure 5.22 shows two example deformations of the Visible Human torso, with the initial object wire shown in the leftmost image. The mapping field for the first deformation (middle image) took 0.23 seconds, and the second (rightmost image) 0.26 seconds. In both cases, the interaction speed after mapping field generation was found to be approximately 6FPS.

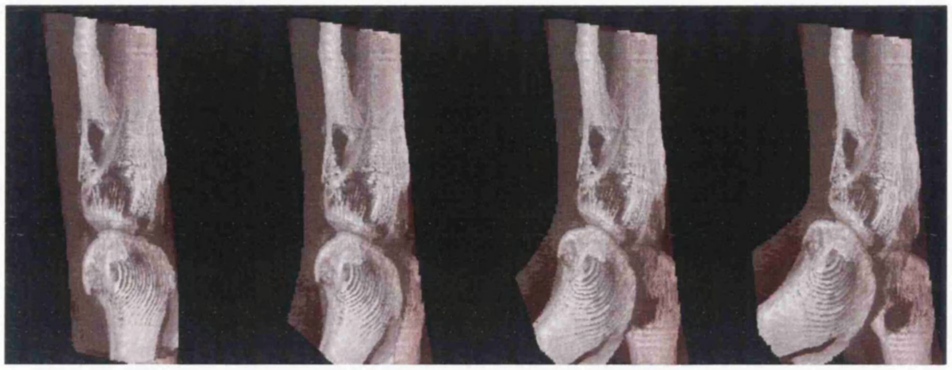


Figure 5.20: Four frames from an animation of the CT Knee dataset

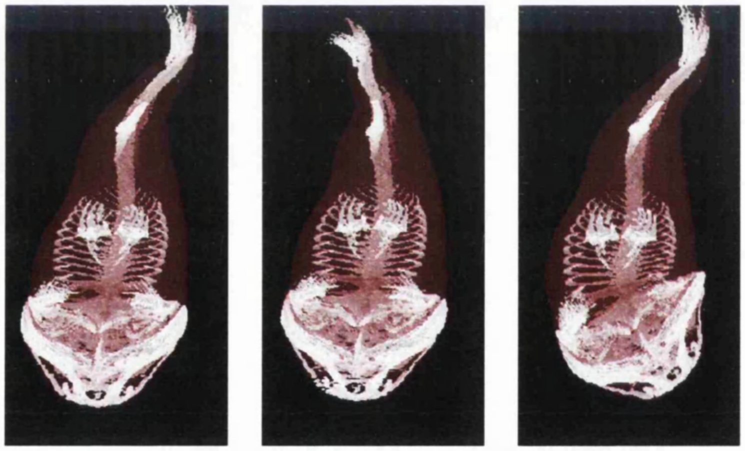


Figure 5.21: Three frames from an animation of the CT Carp dataset

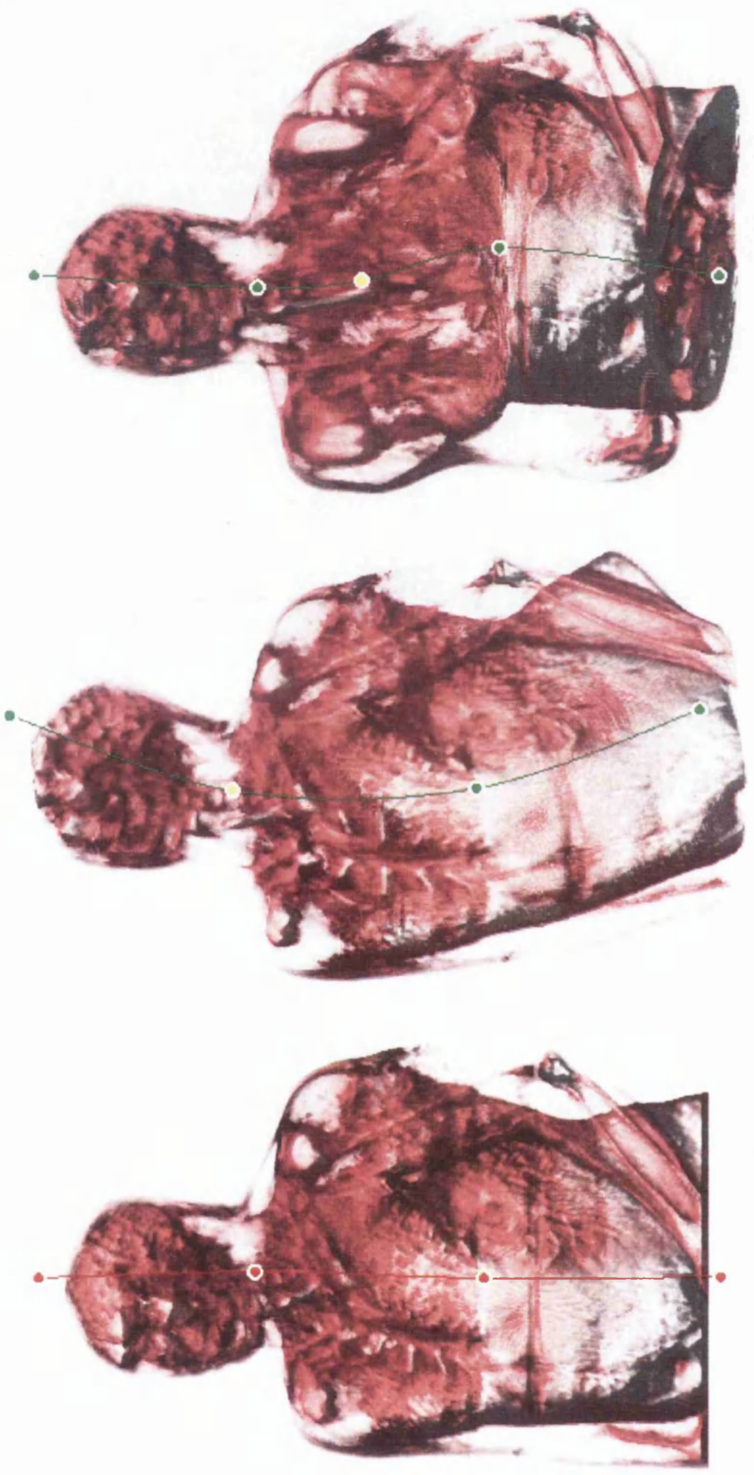


Figure 5.22: Deforming the Visible Human Torso – the leftmost image shows the object wire (red) defined inside the body; the centre and rightmost image show two deformations of the world wire (green) and their final rendering.

5.8 Summary

A volumetric deformation methodology named Volume Wires has been introduced in this chapter along with a group of techniques and algorithms for implementing it. The methodology presented provides an intuitive interface to the user, providing the ability to treat the volume objects as bendable objects attached to *wires* which the user defines inside the dataset. Along with effects such as compression, stretching, and bending, the user can specify explicit effects such as warping and twisting along the wire trajectory.

We have given an analysis of the mapping that exists between object space and world space given a set of user-defined wires, and have introduced a method of encoding this mapping into a volume dataset called the *mapping field*. The mapping field is created using a distance or vector propagation technique, and we have given a visual and analytical comparison of various mapping field generation methods defined for a given deformation.

A GPU-based raycasting algorithm was finally introduced in this chapter that provides an effective means of generating interactive images from a given deformation in the Volume Wires methodology. The algorithm evaluates a given mapping field encoding to generate new sample points in the volume dataset, and thus provides interactive frame rates for a given deformation for varying viewing parameters.

Chapter 6

Forward-Projection of Volume Wires on GPU

Contents

The Volume Wires methodology defines a group of methods and algorithms to perform nonlinear deformation of volumetric data. To validate the effectiveness of the methodology in use however, a more interactive approach that allows the user to perform the deformation interactively is needed. In this chapter, a forward-projection system for the forward-mapping of Volume Wires deformations is introduced in the form of an easy-to-use software tool. The tool is flexible, stable, and can be ported easily to any OS supporting Trolltech's QT toolkit, OpenGL, and NVidia's drivers.

Parts of the work in this chapter has been published in the Proceedings of the Fourth International Conference on Medical Information Visualisation – BioMedical Visualisation (MediViz 2007) [WJ07].

6.1 Introduction

In designing an intuitive and comprehensive user interface for an implementation of the Volume Wires framework, it is necessary to implement suitable visualisation algorithms to enable the user to view the deformation interactively (or at least obtain a *preview*, perhaps using a scheme such as progressive refinement). *Interactively* in this case, we define to be five frames per second or above.

An early implementation of the Volume Wires framework used a simple user interface built with GTK 2 (Gimp ToolKit, a user interface library) on GNU/Linux. The interface presented two purely static views of the volume dataset and allowed the user to define object and world wires. The user would then click a button to produce a raycasted image with the *full mapping process running on the CPU.* A later implementation (the implementation provided in the latter half of Chapter 5) extended this by providing an interactive raycasted image of the dataset (using the GPU) during wire specification. However, there are still two problems with this approach:

- The lack of deformation preview meant that the user had to predict the effect that moving the object and world wires would have on the volume object;
- The large delay that presented itself to the user when the mapping field is built for the current deformation is frustrating.

It is clear therefore that in order to verify the usefulness and intuitiveness of the Volume Wires deformation methodology, an interactive implementation is required that enables the user to deform a given volume object interactively. In the context of the Volume Wires methodology, such an implementation should enable the user to deform the world wires of the object and view the resulting deformation interactively; a limited amount of pre-processing is acceptable if kept to a minimum and not computed while the user is actively deforming a world wire or adjusting the viewing parameters.

As will be discussed in the coming sections of this chapter, the pre-processing operations are limited to the deformation of the object wire defined inside the target object; specifically, a mapping field is constructed for the object wires that permits the deformation in world space. The construction of such a mapping field can be justified in an interactive application as it is only required when the object wires are defined or modified; this is an operation that takes place rarely in the Volume Wires methodology.

The methods introduced in this chapter are targeted specifically at the generation of graphics hardware available at the time of implementation. As the technology progresses, newer possibilities of implementation will arise; this will inevitably make the methods given in this chapter non-optimal. However, such methods will still form the basis of future implementations as the boundary between the CPU and GPU is expected to become fuzzier over time, with architectures such as CUDA [CUD] lifting the sometimes harsh restrictions placed on algorithm developers. This will bring the possibility of new feature-specific optimisations for the methods to further refine the speed and image quality.

6.2 GPU Deformation Strategies

Section 3.4.2 gives an overview of forward and backward mapping functions in the context of the Spatial Transfer Functions framework (Chen *et al.* [CSW$^+$03]). Forward-mapping functions apply the deformation function to each element in object space (in volume graphics, the elements are the voxels) and then render these elements in their new positions in world space. Backward-mapping functions achieve the reverse, discovering sample points in object space based on current world space positions. For the latter case, because there is no guarantee where the samples will fall in object space, the object-space representation must be treated as continuous.

Forward-mapping functions cannot be trivially applied to volume rendering, as the ultimate goal of volume rendering is to produce a *continuous* rendering from a set of discrete samples. As discussed in Section 3.4.2 however, forward-projection methods such as Splatting [Wes90] can achieve visually impressive approximations to this by projecting the individual voxels to the screen. In addition, many GPU-based implementations exist for Splatting, providing interactive framerates by exploiting the blending capabilities of the hardware; though many of the Splatting algorithms available are geared towards point-based rendering applications and thus are not required to offer full blending capabilities (see Section 4.4.3 on *Visibility Splatting*).

Backward-mapping functions are more suited to raycasting volume rendering generally, providing relatively simple image-based methods for revealing the internal texture information of the volume dataset. When the user introduces a deformation however, the rendering algorithm must be able to apply the inverse of the deformation to each world sample point (and additionally bring the computed normal into world space). However, forward-projection provides an attractive property when deformation is added to the pipeline in that only the deformation function Φ is required, not the inverse.

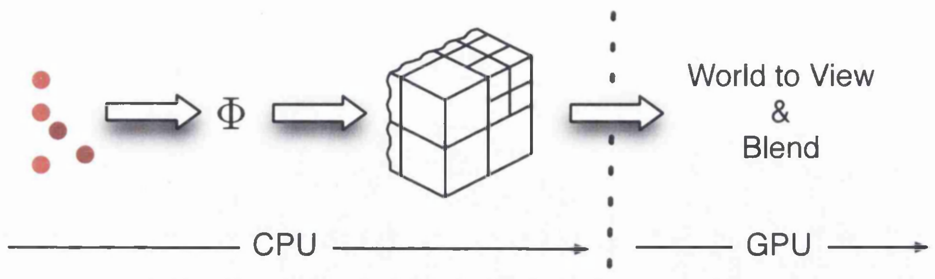
6.2.1 Order of Operations

The principle behind point-based rendering on the GPU is to define each point sample / voxel as a vertex with associated attributes such as colour, opacity and its normal, and then project each vertex to the framebuffer. Special considerations are required for pixels in the framebuffer being affected by more than one splat, as the order of blending determines the final pixel colour. With a CPU-based renderer, the developer of the algorithm has the option of building a depth-sorted list of splats per pixel and then performing the final shading calculations on each list once the projections have been completed. However, the GPU offers no capabilities for storing arbitrary lists in each pixel; though some algorithms such as the k-buffer [CC05] for assisted visibility sorting are able to function by storing very small lists in off-screen buffers.

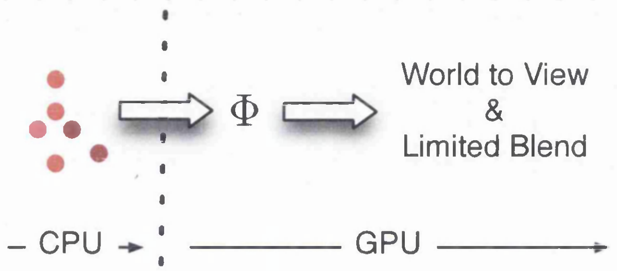
The options for blending are:

- *Depth-sort before projection* – The world-space voxels (vertices) can be depth-sorted on the CPU using an octree before being sent into the pipeline. This will guarantee a front-to-back or back-to-front rendering that is necessary for correct blending operations;

- *Perform limited blending* – Only perform a small amount of blending for the voxels at the very front of all the others, reducing the possibility of depth artefacts but providing a reduced quality image with little internal texture information if such information is available.



(a) Less optimal: the CPU has to perform Φ, but is able to perform depth sorting.



(b) More optimal: The GPU carries out all of the computation, but is unable to depth-sort.

Figure 6.1: Objectives of a GPU forward-mapping function Φ

The first option is suitable only when the data is static – that is, an octree can be computed easily and stored for use throughout the rendering process. Now assume some the existence of some deformation function $\Phi : \mathbb{E}^3 \rightarrow \mathbb{E}^3$ that takes each point sample and brings it into world space based on some user-defined deformation. In order to render the deformation on the point-based dataset, Φ would need to be performed on each and every point sample, and the new point sample positions used in subsequent pipeline operations. This operation can be potentially costly by itself, but ascertaining a correct depth-sort each time the samples are deformed would further exacerbate the cost. In the case of an octree, the octree would need to be recalculated once the deformations had occurred. This would normally not be a problem – intuitively, the cost of building an octree of a set of points is less costly than performing some deformation on each point. It does however limit the possibilities for GPU acceleration, as illustrated in Figure 6.1 where two possible CPU / GPU boundaries are shown.

Figure 6.1(a) shows the situation discussed in the previous paragraph, where Φ is applied to dataset points on the CPU, and then a depth-sort is performed (again on the CPU), before letting the GPU project and blend the newly-positioned samples to the framebuffer. It is clear that it would be extremely beneficial to leverage the parallel processing capabilities of the GPU to perform Φ on each incoming sample. This can be achieved by performing Φ in the vertex shader and modifying the vertex's position semantic. The potential speedup is dramatic, as not only is the GPU is heavily optimised for trigonometric and computationally intensive calculations, but the GPU operates on many vertices in parallel. Once each vertex is modified, it is broken down into fragments by the rasterisation unit, where each fragment is then processed by the fragment shader.

6.2.2 Crack-Filling

One of the major issues with point-based rendering is that of 'filling' the gaps between samples that appear in image space. These gaps occur when neighbouring samples in world space do not map to neighbouring samples in image space, producing a discontinuity and consequently a noticeable gap artefact in the final image. When rendering rectilinear structured volume datasets using a point-based algorithm such as Splatting, this issue is not as common or indeed difficult to deal with because there is some level of guarantee of the space between voxels, and thus suitable fixed solutions can be inferred relatively simply. However, with unstructured point-based datasets obtained from scanning equipment, no such guarantee exists, and thus methods must exist to close the gaps between samples. Some approaches resort to creating new samples in world space, and other approaches work in image space to detect and close the gaps.

Adding deformation to this scenario makes any attempts at closing the gaps much more difficult, as any such attempts must not only deal with potentially large deformations (large gaps would be created as samples are pulled apart), but also must be efficient to maintain high levels of interactivity. Point-based rendering algorithms typically do not support dynamic deformations and can therefore assume a reasonable sampling density. Where such a density does not exist, the image-space splat shape and/or sizes are modified to close any holes in the final image. The PointShop 3D system [ZPKG02] detects insufficient sampling density due to dynamic stretching deformations and introduces new sample points to maintain a continuous surface.

6.3 Method Overview

This section introduces a set of methods and algorithms for the forward-projection of Volume Wires deformations, implemented on the GPU. A diagram illustrating the overall method is given in Figure 6.2.

A two-pass visibility splatting rendering algorithm (visibility splatting is discussed in Section 4.4.3) is used to give an interactive point-based rendering of the deformation as the user is modifying the world wires or changing the viewing parameters. The vertex shader takes in voxel samples encoded as vertices and performs the Volume Wires deformation mapping function Φ on their positions to bring them into world space. The fragment shader colours the resulting fragments based on the data from the volume dataset and the associated transfer function.

In the first pass of the algorithm, the vertex shader shifts the voxel's resulting image-space position (after applying Φ and the ModelView matrix) backwards by ϵ in view space after performing Φ on the voxel's position, generating the required shifted depth-buffer for visibility splatting. In the second pass, the vertex shader performs only Φ, and the fragment shader performs the shading and blending operations on the samples that survived the shifted depth buffer; these samples are said to be within the ϵ boundary. The volume dataset is encoded as a 3D texture in order that the normals can be computed efficiently in the fragment shader. The first pass is typically highly efficient on Geforce 6 hardware (and upwards) due to in-

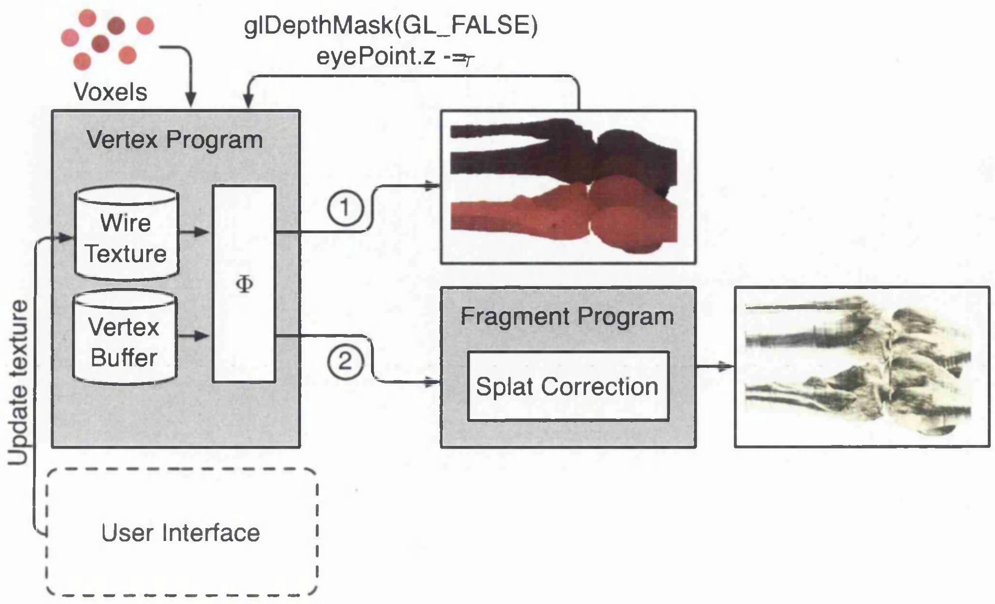


Figure 6.2: GPU Forward-Projection algorithm for Volume Wires

ternal optimisations when writing only to the Z-buffer (known as *double-speed Z-writing* [NVI05]).

The implementation of the deformation algorithm on the GPU is spanned over three main shaders, which will be referred to by name in this chapter:

- *VW_Vertex_Pass1* – The first vertex stage; applies Φ on the voxel and then shifts the resulting position backward by ϵ
- *VW_Vertex_Pass2* – The second vertex stage; applies Φ on the voxel and pre-computes some shading variables for the fragment shader
- *VW_Fragment* – The fragment stage; performs the shading computations on the fragments created by each voxel.

In addition, the system utilises versions of these shaders that have the deformation-specific computations removed. This allows for fast rendering of the volume dataset while the user is initially placing the wires in the scene; branching operations inside the shaders can in theory provide this functionality without the necessity of writing additional shaders, but it was found in practice that the branching overhead was prohibitive on the hardware used for implementation.

6.3.1 Designing a GPU-based Φ

In the backward-mapping function described for the raycasting rendering algorithm in Section 5.7, the closest world wire point is discovered from the data encoded in the mapping field. Conversely, the forward-mapping function must discover, for each voxel, the closest object wire point and apply the forward-mapped deformation function Φ to that voxel. The most suitable manner in which to achieve this would be to encode the closest wire γ and closest parametric offset t with each voxel; once this voxel enters the mapping function Φ, it will be immediately clear how to transform it based on the encoded attributes and the wire data.

In the backward-mapping renderer, a mapping field was created using distance propagation to discover the closest world wire point for each ray sample point. The mapping field can be used in the forward-mapping case also, by using it to associate voxels in object space with their closest wire offset. Instead of creating the mapping field based on the set of all world wires, the field is created based on the set of all object wires. This enables each voxel in the volume dataset to be associated with its nearest object wire point (assuming the voxels can be tagged with additional attributes) by traversing each voxel in the mapping field and tagging its associated volume dataset voxel (based on its position) with the (γ, t) values at the same location.

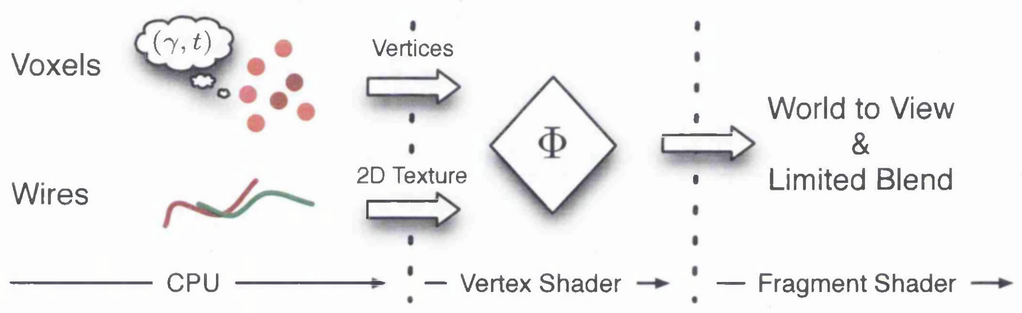


Figure 6.3: Proposed GPU Implementation of a Forward-Mapping Volume Wires Rendering Algorithm

Figure 6.3 gives an overview of the proposed GPU implementation for the forward-projection of Volume Wires deformations. The computation of Φ is performed by a vertex shader running on the GPU. Each voxel in the system is defined as a single vertex with the closest wire and offset attributes encoded into the vertex's colour value: γ can be encoded into the red component, and t into the green component. These values are obtained from the mapping field created based on the set of all object wires. The wires themselves in the scene are discretised into a 2D texture for input to the mapping equation.

The advantage of using forward-projection for the Volume Wires mapping is immediately clear. Since the object wires are designed to represent the static curve-skeletons in the dataset, they are modified infrequently by the user. This in turn implies that the mapping field will be constructed infrequently during the deformation process – since the vast majority of interaction with the system is carried out by the user modifying the world wires. When the world wire is modified, each voxel being brought into world space using the closest object wire point as input to Φ. This makes a forward-projection system for the Volume Wires methodology extremely attractive from a usability point of view.

In a typical volume dataset, only a subset of the voxels can be classified by the user as non-background, or *active* voxels. For example, in a CT dataset, the voxels with small density values (perhaps representing the surrounding air or some form of packing material) are not useful in the sense that they add value to the visualisation. The distinction between background and active voxels is of small importance in backwards-projection renderers. Such values are typically removed from visualisations by setting a transfer function that sets voxels with these values as having zero opacity. However, in forward-projection rendering where each sample must be represented by a set of attributes (position, colour, etc) and deformed individually, this distinction is of utmost importance for efficient rendering. Therefore, all operations on voxels in the system given in the coming sections are performed on voxels deemed *active* by the user – achieved by setting a single scalar threshold value β which separates background from active voxels by discarding those with values falling below it.

6.3.2 Wire Encoding

The ability to fetch random-access data in the vertex shader (with runtime array indices) can be obtained using the Vertex Texture Fetch (VTF) functionality of modern graphics hardware (ShaderModel 3.0). The ability to fetch texture data in the vertex shader is a relatively recent feature for GPUs, as texture fetches are most intuitively required in the fragment processing stage to apply texture data to the fragments created from primitive objects. However, vertex texture fetches allow for a great degree of flexibility by allowing the vertex shader to modify the geometry of the scene based on texture data [GFG04]. The system introduced in this chapter uses VTF functionality to obtain the wire points from a texture, which are then given the forward-mapping function to complete the mapping.

The internal format chosen for the wire texture is *GL_RGBA_FLOAT32_ATI* and *GL_RGB_FLOAT32_ATI* for all others. These texture formats offer full 32-bit components (128bit per texel for RGBA), and have the added advantage that they are unclamped formats, and thus will not be subjected clamping to $[0, 1]$ by OpenGL prior to GPU upload. In addition, the formats are one of few internal formats allowed for vertex texture fetches at the time of writing [GFG04]. Another benefit of this format is that unlike with some unclamped, 32-bit formats, it does not necessitate the use of extension-based texture targets, such as the *texture_rectangle* range of extensions. These texture targets allow for non-power-of-two textures, at the expense of reduced efficiency with texture lookups at runtime. In addition, these texture targets use integer-based addressing; as will be discussed in coming sections, this would be less than ideal.

6.3.3 Vertex/Voxel Data Encoding and Upload

Each active voxel in the scene is encoded as a single vertex with the associated (γ, t) (obtained from the mapping field) encoded into the colour value. These vertices are stored as two floating-point buffers and uploaded to the GPU using Vertex Buffer Objects (VBOs). An overview of Vertex Buffer Objects has already been given in Section 4.4. VBOs allow for large amounts of vertex and vertex attribute data to be stored and evaluated on the GPU. Example data includes colour (which is used by this system to store the (γ, t) attributes), secondary colour, normal, and texture coordinate. Each of these attributes is contained within a separate buffer; they are not interleaved.

The system given in this chapter uses the vertex (position) and colour buffers to store the voxel attributes. Table 6.1 gives the VBO arrangement for the system:

Attribute description	**Type**	**Buffer Used**
Voxel XYZ position in object space	3 floats	Vertex Buffer
(γ, t) pair for the voxel	2 floats	Colour Buffer
'Deformed' flag - set to 0 or 1	1 float	

Table 6.1: Vertex Buffer arrangement

The total memory required for the buffers on both the CPU and GPU (since each ultimately has a copy) depends on the number of active voxels in the dataset. For example: a dataset containing 4,000,000 active voxels requires $4,000,000 * (3 + 2 + 1) * sizeof(GLfloat) \approx$ 92 MB. *Note: that the voxel normal data is computed at runtime in the fragment shader rather than being bundled with the voxels, saving 12 bytes per voxel.*

The position buffer stores the voxel's position in object space and is required to be uploaded only upon loading a new volume dataset. The colour buffer is used for the (γ, t) attributes for each voxel (the data for which is obtained from the mapping field created from the object wires, as discussed in Section 6.3.1) and needs to be recomputed and uploaded upon modification of an object wire. In addition, the blue component of the colour buffer is used as a flag to denote whether the voxel must be translated to a new position with Φ; voxels set to 0 are unmodified and will be simply projected to the screen in their object-space positions. Such a flag allows for parts of the dataset to be singled out for deformation based on some criteria decided on the CPU.

As discussed in Chapter 5, a mapping field need not be the same size as the volume dataset. The choice of size of the mapping field is a trade-off between the time taken to generate it, and the quality of the mapping. The same principle applies in the forward-projection case, since the mapping field is required to be computed before the user can proceed with deforming the volume dataset by manipulating the world wires.

Table 6.2 gives the amount of time to generate the mapping field for two datasets, along with the resulting size of the field for three chosen dataset to field ratios. Additionally, we give the time to modify and upload the VBO vertex/color buffer with the new attributes once the mapping field has been recreated.

Dataset	Field size	Generation time	VBO modification time
Visman Torso	$77 \times 47 \times 100$	0.1s	Negligible
	$155 \times 95 \times 200$	0.76s	0.29s
	$310 \times 190 \times 400$	6.16s	0.56s
CT Carp	$64 \times 64 \times 128$	0.14s	Negligible
	$128 \times 128 \times 256$	1.10s	0.31s
	$256 \times 256 \times 512$	8.93s	0.71s

Table 6.2: Mapping field generation timings

6.4 Rendering Algorithm

Once the required buffers on the CPU have been computed, organised and uploaded to the GPU, the GPU is able to begin rendering frames. The rendering of a frame is instantiated by either the user changing the viewing parameters (i.e. rotating the volume dataset), or deforming a world wire. In either case, the entire deformation must be computed from the object-space voxels, since the vertex position states cannot be saved between frames in the case of changing the viewing parameters.

Algorithm 3 Forward-Projection Rendering Algorithm Pseudo-Code : CPU Side

```
 1: {Pass one}
 2: Depth : 1, DepthFunc : <, DepthMask : 1
 3: Bind VW_Vertex_Pass1
 4: Rasterise voxels in buffer
 5: {Pass two}
 6: Enable point sprites
 7: DepthMask : 0
 8: Bind VW_Vertex_Pass2
 9: Bind VW_Fragment
10: Rasterise voxels in buffer
```

Algorithm 3 gives the pseudo-code for the most important steps of the CPU side of the rendering algorithm. The first pass runs only vertex program *VW_Vertex_Pass1* and writes to the depth buffer only (the disabling/enabling of colour writes is omitted in the pseudo-code for clarity). After multiplying the new vertex position by the modelview matrix, the shader shifts the new voxel position in image-space backwards by ϵ so that in the next pass, only fragments in front of this boundary will pass the depth test. The second pass runs vertex program *VW_Vertex_Pass2* to compute Φ along with some important shading variables for fragment program *VW_Fragment*, which computes the final colour of the voxels closest to the viewer inside the ϵ boundary defined by the visibility splatting procedure.

The next sections detail the GPU shader implementation of the rendering algorithm – Section 6.4.1 details the computation of Φ in the vertex shader; Section 6.4.2 details the computation of the final voxel colour in the fragment shader; and finally Section 6.4.3 details a method for closing the gaps that appear between image-space samples that utilises the vertex and fragment shaders.

6.4.1 Performing Φ in the Vertex Shader

The vertex shaders *VW_Vertex_Pass1* and *VW_Vertex_Pass2* must transform each voxel first from object to world space, and then from world to view space; the latter is the most fundamental operation performed by the vertex shader and is achieved by a multiplication with the modelview projection matrix. The former operation is performed specifically for the Volume Wires deformation, and is discussed below.

To perform Φ on the voxel, the (γ, t) pair are retrieved from the voxel's colour values (encoded into the colour values from the mapping field on the CPU) and used as offsets within the encoded wire textures to obtain the relevant wire information. The manner in which the wire data is obtained and computed is the same as in the backward-mapping case, given in Section 5.7.7.

Since the combined mapping matrix contains redundancy, the mapping is performed in smaller stages, as shown in the Cg Listing 6.1 :

Listing 6.1: Computation of Φ in the vertex shader

```
1  half row = 1.0f / wireCount;      // height of wire texture row in [0,1]
2  half col = 1.0f / wirePrecision; // width of a column
3
4  // only deform voxels with 'deformed' flag set
5  if(voxel.b == 1) {
6     half wireID = voxel.r;
7     half t = voxel.g;
8
9     // fetch the wire points, trajectories, and normals
10    float3 P_o = tex2D(wire,half2(t,wireID);
11    float3 P_w = tex2D(wire,half2(t,wireID+row));
12    fixed3 T_o = normalize(tex2D(wire,half2(t+col,wireID))-P_o);
13    fixed3 T_w = normalize(tex2D(wire,half2(t+col,wireID+row))-P_w);
14    fixed3 N_o = tex2D(normals,half2(t,wireID));
15    fixed3 N_w = tex2D(normals,half2(t,wireID+row));
16
17    // calculate the binormals (N x T)
18    half3 B_o = normalize(cross(N_o,T_o));
19    half3 B_w = normalize(cross(N_w,T_w));
20
21    // perform the forward-mapping on the vertex
22    vertex -= P_o;
23    vertex = mul(half3x3(B_o,N_o,T_o),vertex);
24    vertex = B_w*vertex.x+N_w*vertex.y+T_w*vertex.z;
25    vertex += P_w;
26 }
```

Lines 6 and 7 create the required binormals from the normal and tangent vectors obtained from the textures. The mapping takes advantage of hardware matrix multiplication for the first stage (line 23) since the matrix can be defined neatly inline in column order. The second stage of the mapping (line 24) takes advantage of vectorised multiplication, equivalent to transposing a 3x3 matrix created from the B_w,N_w and T_w vectors. The final step of the vertex program (not shown for brevity) is to multiply the new vertex position by the modelview matrix.

The vertex texture fetch functionality typically incurs a higher latency than the equivalent fragment texture fetches. This is particularly true of NVIDIA cards such as the 6 series, where vertex texture fetches suffered from larger latency than their fragment counterparts due to the fact that the functionality was brand new and mostly unoptimised. It has been verified from the native assembly code generated by NVIDIA's Cg compiler that part of the latency of fetching the texture information is hidden automatically by the Cg compiler by performing the first half of the mapping as the object data is available.

Attribute	Description
TEXCOORD0	Original object-space position (before Φ applied)
TEXCOORD1	Point Sprite Coordinate
TEXCOORD2	Image-Space T_w
TEXCOORD3	N_o
TEXCOORD4	N_w
TEXCOORD5	T_o
TEXCOORD6	T_w

Table 6.3: Texture coordinates utilised in passing data from the second vertex stage to the fragment shader.

6.4.2 Shading in the Fragment Shader

The fragment program *VW_Fragment* runs only on the second pass of the algorithm, and therefore operates only on voxels within the ϵ boundary generated by the depth-shifting operations.

For lighting calculations, the voxel normal must first be calculated from the volume dataset 3D texture and brought from object to world space – the computation of the voxel normal is therefore performed *per-fragment* instead of per-vertex. Although there is a certain amount of redundancy in performing the normal calculation for each fragment (each vertex will create multiple fragments, all with the same normal), the alternative is to precalculate the voxel normals on the CPU and store them in a VBO of their own. This would present a unnecessary memory overhead since each voxel normal would consume 12 bytes of GPU memory and potentially millions of active voxels are defined in the dataset. The vertex shader cannot calculate the normal from the volume dataset since vertex shaders currently have only access to 2D textures, although this situation is expected to improve with future implementations of the vertex texture fetch functionality.

The fragment shader needs to have access to the original object-space position of the incoming fragment to calculate the normal of the voxel it is representing. Since the Φ has been performed on the incoming fragment, its original object space position can no longer be inferred using the inverse of the original modelview projection transformation. The second pass of the vertex shader therefore writes the original object-space position to one of the outgoing texture coordinates. The central differences operation can now be performed using this texture coordinate to give the object-space normal for the current fragment.

The final stage of the fragment shader must perform Φ_n on the object-space normal to bring the normal into world space based on the current deformation. Such a transformation requires access to the object and world normals / trajectories N_o, N_w, T_o, and T_w, but not the object and world wire points P_o and P_w due to the fact that the translations are not included in Φ_n (unlike with the standard Φ). These variables could potentially be obtained from the textures as with the vertex stage, but a more efficient solution would be to encode these variables into the texture coordinates available for general use to pass data from the vertex to the fragment stage. Such coordinates are available to be written to by the vertex program and read from by the fragment program, and are primarily used for applications such as multitexturing where layers of textures are blended together to form a final fragment colour.

Table 6.3 shows which texture coordinates are used by the algorithm. The required object and world normals / trajectories N_o, N_w, T_o, and T_w are encoded into the texture coordinates by the second vertex stage, which are subsequently available to the fragment shader for each fragment created from the voxel. The final voxel normal is computed using the normal-specific mapping Φ_n. The voxel colour is computed from a transfer function encoded as a 1D texture using the grey-level value obtained from the volume dataset. Lighting is then applied to the fragment using the Cg function `lit` to compute specular and diffuse coefficients.

6.4.3 Closing Cracks along the Wire Trajectory

During deformation of a discretely sampled object such as a volume dataset, two types of crack artefacts may be introduced – those that correspond to a joint in the curve skeleton, and those due to the expansion of a deformed region. The former problem has been encountered previously by Silver *et al.* [GS01, SSC03] and one solution is that of using mid-plane geometry [SSC03] to artificially stretch and warp the individual the skeletal segments to fit. This solution however may not produce a smooth connection, as it is purely linear in nature.

With regards to GPU point-based rendering, creating new samples in the vertex buffer would be costly since samples would need to be added on the CPU and then re-uploaded. The dynamic creation of vertices on the GPU is now possible using geometry shaders, although at the time this research was conducted, such functionality was not available. A highly limited form of vertex creation could be realised however by rendering to a texture and reinterpreting this texture as a vertex buffer during subsequent passes (known as *render-to-texture*). This method however would be impossible with even a reduced-sized volume dataset, since the maximum 2D texture size would restrict the number of voxels.

The Volume Wires system allows for the sharp bending of objects, and stretching and compression of the volume object along the wire as a by-product of the deformation process. A backward-mapping algorithm can recover values in these areas simply by interpolation in the original volume dataset. However, when using a forward-projection algorithm with a finite set of voxels, cracks can occur in image space as the voxels are pulled apart along the wire trajectory (see Figure 6.4). This effect is also apparent along the outer edges of a bent object where the curvature of the wire is high. The effect of the gaps in the image is often distracting and disconcerting, and additionally spoils the aesthetic quality of the final image. It would be highly beneficial if the forward-projection algorithm for the Volume Wires methodology to cover these cracks as much as possible to maintain a smooth, continuous image without revealing the true discrete nature of the underlying data to the user.

An observation of the Volume Wires methodology is that the 'stretching' of samples (and subsequently, the gaps the occur) occurs along the image-space trajectory of the wire. If the samples could be stretched in image space along the trajectory of the wire, then these gaps could be reduced substantially, greatly improving the image quality. This process would require that the shape of the voxels processed by the fragment shader are altered, and is a common operation in point-based rendering algorithms whereby the splat shapes are altered depending on their normals such that the resulting shape is a disc. This shape-changing ability is achieved by obtaining an image-space parameterisation of each point such that each fragment is aware of its image-space position relative to the original point. The parameterisation functionality is given by the `point_sprite_nv` OpenGL extension. When enabled, the extension forces the rasterisation unit to break point primitives into squares represented by $n \times n$ fragments, where n is the current point size set in OpenGL. Associated with each fragment is a texture coordinate in $[0, 1]$ representing its position within the square; this coordinate can be used to govern the shape of the final point by selectively discarding (or alpha-modifying) fragments based on the texture coordinate.

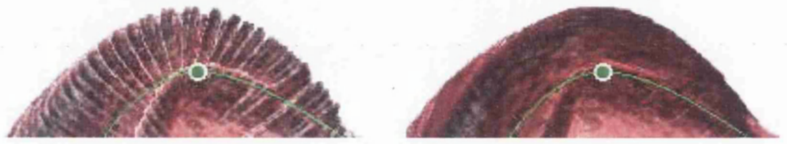


Figure 6.4: Correcting cracks

To 'stretch' the shape of point samples along the wire trajectory, it is necessary to set a large enough point size to give a large number of potential fragments. Therefore, the point size and the splat size become separate variables in the system; the former governs the number of fragments that a point is broken into, and the latter governs the number of these fragments accepted through the pipeline to give the final size of the sample (without regard to the stretching). The point size chosen is a tradeoff between the effectiveness of the splat correction (larger points give a larger space for splat correction) and speed (bigger points produce a larger number of fragments which must be processed by the fragment shader).

As the user modifies the world wire, the distance between consecutive points on the world wire is compared to the points on the corresponding object wire to obtain a scaling value s, which is stored in the α component of the object wire texture for each point (the α component of the world wire texture is already used by the scaling effect values). s is derived for each wire point as the distance between the current control point and its previous point over the distance between the current object point and its previous point. In the second vertex stage, the vertex shader computes the image-space wire tangent from T_w by multiplying it with the inverse transpose of the modelview matrix, storing the result in a texture coordinate for input to the fragment shader.

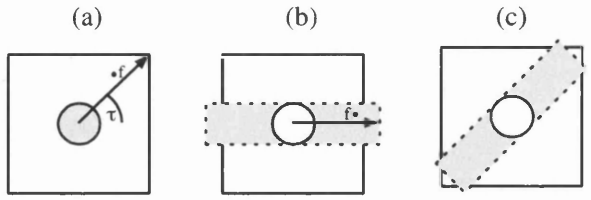


Figure 6.5: Correcting the splat shape along wire trajectory

The fragment shader must now make the decision to either accept (shade) or reject (discard) the fragment based on this information. This decision is made by testing whether the fragment lies within a rectangle of $splatSize$ height defined around the image-space tangent vector T. The position of the current fragment f within the point sprite is obtained from the texture coordinate given by the point sprite extension. First, the fragment f is brought into the tangent's space:

$$\begin{bmatrix} T_x & -Ty \\ T_y & T_x \end{bmatrix} \cdot (f - 0.5)$$

The test for fragment retention is now reduced to a simple inside-rectangle test, with the splat size used to determine the y-extent of the test. In the example shown in Figure 6.5, fragment f survives the test and continues to be processed. Any fragments failing the test are destroyed using the `discard` keyword.

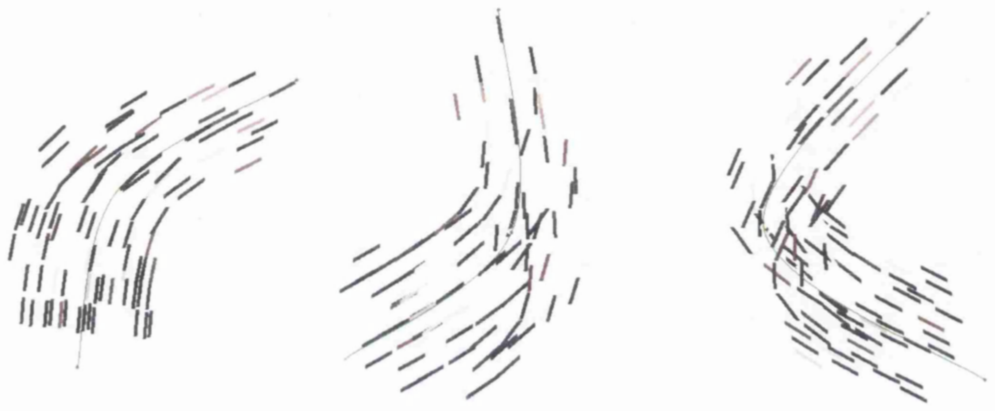


Figure 6.6: Splat Shape Correction – Exaggerated Rendering

The effect of the splat shape correction in this manner is shown in Figure 6.6, where a small subset of voxels have been chosen with an exaggerated size and opacity to better show the effect.

6.4.4 Progressive Refinement

For large volume datasets with many active voxels, a significant drop in the number of frames per second can occur during interaction with the wires. This is due to the high number of computations being performed in the vertex and fragment shaders. This slowdown will be particularly noticeable on lower-end graphics cards which suffer not only from fewer vertex and fragment shaders, but also from slower memory speeds and less onboard RAM to store the vertex buffers. Unavailability of a particular feature of the storage area often entails a CPU emulation of that method if feasible, which results in vastly reduced performance.

To speed up a system based on forward-projection, the number of samples that define the volume object must be reduced, and the splat size increased to compensate for the larger gaps between samples; analogous to reducing the size of an image and zooming in such that the viewable size is the same. This technique however often leads to significant loss of image quality, as the fine details of the volume object are lost; the volume data is given a smoothed out appearance as if a low-pass filter has been applied.

A suitable solution to the general problem of low interactivity, adopted by many interactive graphics applications, is to use an progressive refinement scheme whereby the user is presented with a low quality version of the object for interaction, and a high-quality version when the interaction stops. Progressive refinement for raycasting volume rendering has already been discussed in Section 2.4.4.

There are three approaches that have been identified for reducing the number of samples that ultimately make it through the pipeline, all of which have trade-offs in terms of efficiency and memory usage, and all of which have been implemented for use with the forward-projection algorithm. The approaches are object-space approaches; operating on the voxels (above the β threshold) in object space before they are transformed into world space.

Discarding Vertices in the Vertex Shader

The first approach is to selectively discard vertices entering the fragment shader such that Φ is never performed and the vertex ultimately creates no fragments. Vertices cannot currently be discarded in the same manner as fragments; however, they can be transformed in such a way that they are guaranteed to lie outside of clip-space, resulting no rasterisation unit invocation and therefore no fragment processing. In addition, with dynamic branching, Φ need never be computed for these voxels as the rejection is performed in object-space before Φ is applied. The sparse set of voxels can be computed simply using modular arithmetic on the voxel's object space position (assuming the object space is the size of the dataset and not in $[0, 1]$) to select only even coordinates. Cg pseudo-code for such a scheme is given below.

Listing 6.2: Selecting only voxels belonging to the 'sparse' set

```
if(mod(floor(voxel.x),2) == 0 &&
   mod(floor(voxel.y),2) == 0 &&
   mod(floor(voxel.z),2) == 0) {
   // perform forward-mapping on voxel
   voxel = phi(...);
}
else {
   // this voxel will fail to fall in clip-space
   voxel = float4(-9999,-9999,-9999,0);
}
```

From experimental usage, this scheme was found to have a performance hit during rendering of approximately 30% over precomputing the sparse set.

Using a Vertex Indexing Buffer

The second approach is to give the GPU some indication of which voxels to choose to be sent into the pipeline when given a pointer to the voxel buffer. OpenGL allows for arbitrary 'striding' of the data within a vertex buffer on 1D basis only – resulting in severe artefacts when the buffer is ultimately interpreted as a 3D representation. It is possible to organise the data in memory to account for this before uploading; however, it is desirable to maintain the standard memory layout to avoid such re-computation when modifying the voxels on the CPU.

Vertex Indexing is a technique that allows for the creation of an element index array that contains a list of indices to a vertex buffer. This index buffer is bound before drawing the vertices to instruct OpenGL to look up the indices of each vertex in the element index array, rather than sending them all to the graphics card. During the construction of the Vertex Buffer Objects, an additional buffer is constructed to hold the 32-bit integer vertex indices that represent the sparse set of voxels. When the user is actively deforming the world wires, this buffer is used to govern which voxels ultimately are put into the pipeline.

Unfortunately, it was found in practice that the vertex indexing method gave a severe speed hit of around 30% during rendering (a very similar hit to the vertex shader discard method), possibly due to the fact that the vertices are selectively uploaded one-by-one rather than in one large memory block.

Using Two Buffers

If the graphics card memory allows it, two separate buffers can be defined: a dense buffer and a sparse buffer. The buffers can then be selectively switched at runtime simply by instructing OpenGL to use one buffer or the other depending on whether the user is currently deforming a world wire. If both sets of buffers will not fit in GPU memory at the same time, then they will need to be uploaded on demand.

Though this method is by far the most naive, it has been been found in practice to be the most effective at giving an interactive deformation experience for larger datasets.

6.4.5 Performance

We now give an analysis of the performance of the rendering algorithm by deforming a variety of datasets and measuring the average frame rate while actively deforming the dataset over a period of 30 seconds. The implementation of the system was written on a Pentium 4 at 3.4GHz with 2GB RAM, in C++ on GNU/Linux x86 using NVIDIA's Cg tookit and OpenGL. The graphics card used for measurements is a GeForce 8800.

Each dataset entry in Table 6.4 contains two rows with differing voxel values. These values indicate the number of active voxels based on the progressive refinement level; the first entry denotes the number of active voxels present when the user is actively deforming, and the second entry (with a larger number of voxels) represents the full dataset. The latter entry therefore gives an indication of how much of a delay exists between the user releasing the mouse button after deforming a world wire and the refined image being rendered.

Dataset	# Active voxels	Average FPS
Visman Torso	716,139	8.76
	4,112,368	1.68
CT Carp	641,022	8.93
	5,128,313	1.37
Tooth	52,536	30.15
	420,289	13.09

Table 6.4: Forward-Projection timings

6.5 Incorporating Segmentation Information

The segmentation of volume datasets is a useful means of adding semantic information to the dataset for the purposes of data analysis and manipulation. A large variety of segmentation algorithms exist for volume data, each with their own strengths and weaknesses, that are capable of ultimately outputting a volume dataset where each voxel is labelled according to the unique ID of the disjoint segmented 'set' it belongs to. A review of volume segmentation algorithms has been conducted in Section 3.8.

The segmentation of disjoint regions within a volume dataset can be of great benefit to volume deformation algorithms, as it allows the user to modify individual semantic regions of the dataset. It would be of particular advantage to the Volume Wires framework to introduce such semantic information, as the wires are analogous to the skeletons that comprise many character-based deformations and animations; the added advantage being that due to the fact that space is curved around the wires, joints need not be treated as special cases, producing smooth transitions from segment to segment.

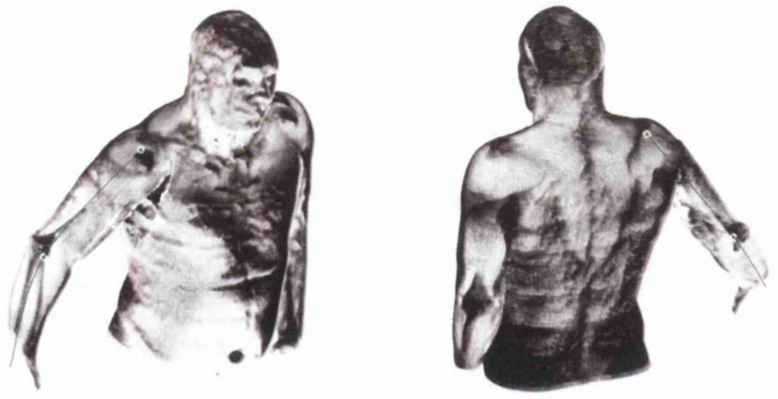


Figure 6.7: Associating segmented portions of the of the volume with each wire

Figure 6.7 shows an image produced from the forward-projection algorithm when segmentation information is incorporated. In this case, the segmentation procedure has marked the right arm of the Visible Human, which has then been associated with an object wire defined within the arm. In the image, the world wire has been moved in such a way as to position the arm away from the body. From an algorithmic point of view, only the voxels that have been associated with the wire have had Φ applied to them to bring them into world space; all other voxels in the scene (those belonging to the head, body and left arm) in this example remain static.

The following sections give details on the manner in which the segmentation information is specified, and how the various stages of the pipeline discussed previously in the chapter are modified to take the segmentation into account.

6.5.1 Segmentation Functionality and Data Format

For the coming discussions, we assume that there is some segmentation method which produces a *mask volume* marking the voxels in the volume dataset belonging to each wire. The mask volume is a signed 8-bit volume dataset of the same dimensions as the target volume dataset, where each voxel contains a value representing the index of the wire it is associated with. The default value for a voxel is -1, meaning that the voxel is not associated with any wire and therefore remains static throughout the deformation process.

A variety of segmentation algorithms have been devised for use with the forward-projection algorithm, including a seed filling algorithm and an algorithm based on energy-minimising Snakes. The latter algorithm has been implemented in a more advanced form into a complete tool, which is discussed in detail in the next chapter.

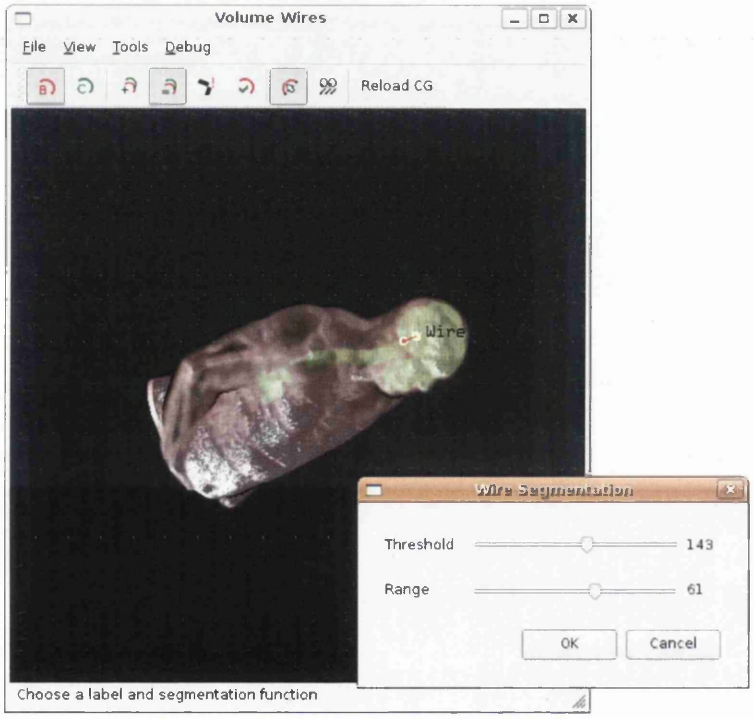
A 3D seed filing algorithm has been implemented with some success in the system. The user interface for the algorithm is shown in Figure 6.8, where the user is presented with a small dialog box to control the seed fill, and a dynamically updated green overlay on the volume dataset to show the segmented regions. The seed points for the filling algorithm are chosen as a small sequence of points along the wire. Two parameters control the seed filling algorithm: a *threshold* and a *range*. The threshold value defines the boundary of the fill (voxels above and equal are filled), and the range value defines the maximum distance that the seed fill algorithm is allowed to operate.

The green overlay on the image is computed by dynamically updating the seed fill volume represented as a 3D texture in GPU memory. The pipeline has one final fragment stage added to the end which fires rays through the mask volume positioned in the same location as the volume dataset. If a value representing a 'filled' voxel is found along the ray, then the final pixel colour is given a green hue; otherwise, the pixel is passed through. This is a similar feedback method to that implemented by Hadwiger *et al.* [HBH03], though without complex blending of the mask volume.

In Figure 6.8, a small wire has been placed inside the head to show which areas of the volume are deemed to be connected to the head by the filling algorithm. It is apparent from this example that the majority of the brain, skull, and some of the spine have been filled. In practice, the seed filling algorithm was found to be rather unreliable and inaccurate for most tasks. For example, attempting to segment the arm of the Visible Human resulted in an inaccurate result due to the fact that the arm of the Visible Human is resting upon his stomach, resulting in a connection when the algorithm hunts for the next candidate point. Methods to avoid this could include:

- Pre-filtering the dataset using Adaptive Nonlinear Diffusion or a similar filter;
- Setting a smaller range parameter for the algorithm;
- A more specialised algorithm that makes use of heuristics to judge when a 'spill' of such a nature exists.

However, it is clear that such solutions are merely skipping around the problem of the segmentation method being fundamentally difficult to control, and segmentation in general being an unsolved problem.



Threshold 143
Range 61

Choose a label and segmentation function

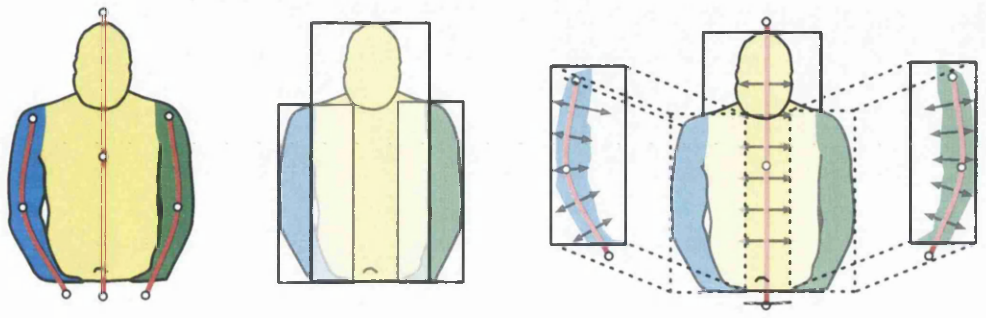
Figure 6.8: Basic seed-filling functionality to create the segmentation mask volume

The segmentation method used to produce the mask volumes in the remainder of this chapter is based on energy-minimising Snakes. The mask volume can be created within the user interface using whatever segmentation functionality is available, or loaded into the system in the same manner as a volume dataset. Although there is a memory overhead in loading the mask volume, it is not required by the system during the deformation process itself, and is never uploaded to the GPU. The mask volume can therefore be discarded before the user begins modifying a world wire.

6.5.2 Segmentation-Aware Mapping Field

When segmentation information is introduced into the system, the task of identifying which wire a particular voxel belongs to is no longer a case of asking which wire is closest; the closest wire may not be the wire that the user has decided should be associated with that voxel. For example, if the user has defined a wire for the Visible Human's right arm and one for his torso, voxels at the edge of the torso may be closer to the arm wire than to the torso wire; and yet the user wishes that such voxels are associated with the torso.

For clarity in the text below, all references to wires are to be taken as the object wires only.



(a) Mask Indexes (b) Bounding Boxes (c) Union of mini mapping fields

Figure 6.9: Building the segmentation-aware mapping field from the union of 'mini' mapping fields

It is clear that the mapping field generation algorithm must be modified to incorporate this segmentation data obtained from the mask volume. The standard mapping field represents, for each voxel, the closest offset on the closest wire. The mapping field that is required for a segmentation-aware rendering algorithm should represent, for each voxel, the closest offset for the *associated* wire based on the segmentation information. Distance propagation methods (such as Chamfer and EVDT as used in the Volume Wires framework) rely on propagating distances globally through the dataset; such algorithms would fail make sense if instructed to propagate only within the associated wire index regions since they require information from their neighbours at each step.

A segmentation-aware mapping field can be realised by creating a 'mini' mapping field for each segmented region, and then pasting the mini mapping field into place in the full mapping field based on the mask volume. Figure 6.9 gives an illustration of the segmentation-aware mapping field generation algorithm. The first step of the algorithm is to create a bounding box for each wire in the scene. For each wire with index i ($wire_i$), the mask volume is scanned for voxels containing value i, and the minimal / maximal coordinates for the union of such voxels are tracked. The result is a bounding box $((x_1, y_1, z_1), (x_2, y_2, z_2))$ for all voxels affected by $wire_i$. Next, a 'mini' mapping field is constructed for each $wire_i$ with the same dimensions as its computed bounding box. This 'mini' mapping field represents the minimal offset t to $wire_i$, and is created by propagating distances from only $wire_i$ to the outer edges of the bounding box. Essentially, it is as if the volume dataset has been cut around the bounding box, and a mapping field created with the remains.

Once this 'mini' mapping field has been created, it is pasted in place into the full mapping field. The pasting procedure is given below:

```
1: $bboxes \leftarrow createBoundingBoxes()$;
```

```
2: for all Wires $Wire_i$ do
3:   for Voxel $v \in$ MiniMapField do
4:     if $MaskVol(v + bboxes[i].origin) == i$ then
5:       $FullMapField(v{+}bboxes[i].origin) = MiniMapField(v{+}bboxes[i].origin)$
6:     end if
7:   end for
8: end for
```

The pasting procedure involves placing the mini mapping field into position (based on its bounding box) inside the full mapping field, and copying each voxel into the volume dataset only where the mask volume value is equal to the wire index. Once this process is completed for all wires, the end result is a mapping field which contains distances propagated to its *associated* wire based on the segmentation information. This mapping field can now be used as input to the procedure for creating the vertex buffers, without any modification to the vertex buffer generation procedure itself since each voxel in the mapping field now correctly contains a reference to its associated wire and the nearest offset on that wire.

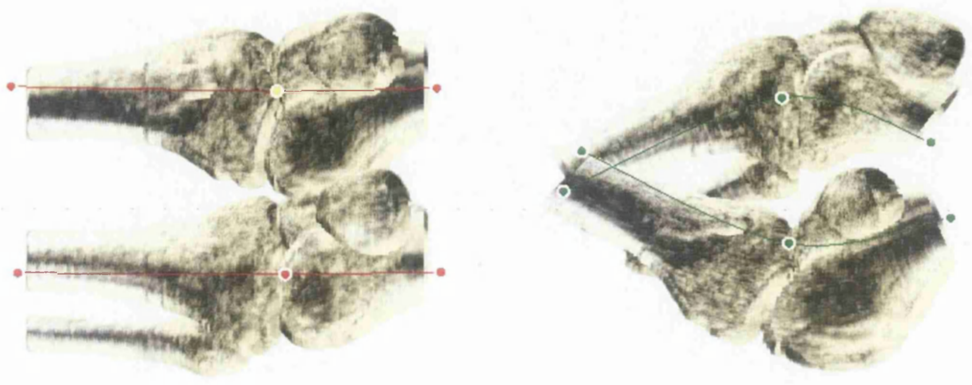
The segmentation-aware mapping field generation process does not add any significant time penalty over the standard mapping field generation process. The biggest overhead in the time to generate the field with respect to a standard mapping field is the amount of overlap that exists between the bounding boxes of each segmented region; such regions will have distance computations performed for more than one wire. In practice, for simple segmentation operations such as segmenting the arms of the Visible Human, we found that incorporating segmentation information often brought the mapping field generation times down due to the fact that the volume of the union of the bounding boxes was far less than that of the full volume dataset.

6.5.3 Continuity at Wire Ends

When segmentation functionality is introduced to the system, care must be taken to ensure continuity at the ends of wires with respect to the other volume data. If the trajectories of the world and object wires do not match at the end of the wire, then a visible discontinuity will exist.

6.6 Result Images

Figure 6.10 gives an example deformation of the CT Knee dataset, with the object wires shown in the leftmost image and the world wires (and resulting deformation) shown in the rightmost image. Figure 6.11 shows an example deformation of the Tooth dataset. Figure 6.12 shows a split of the CT Carp dataset. The split in this case is not explicitly defined – the split is a byproduct of the mapping field process, whereby each voxel will be associated with a closest point on either one wire or the other. When the two world wires are pulled apart, it is clear that the divide occurs at the points halfway between the object wires. Figure 6.13 gives two more example deformations of the CT Knee dataset. It should be noted that in the case of the CT Knee dataset, no explicit segmentation functionality was required to associate each leg with its wire; the gap between the legs was wide enough such that their regions of influence were in-between the legs. Figure 6.14 finally shows four example images of deformations specified on the Visible Human torso dataset.



(a) Object wires (b) World wires

Figure 6.10: Deforming the CT Knee dataset ($\approx$ 15 FPS)

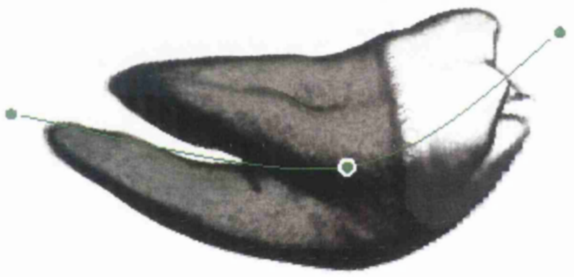


Figure 6.11: Deforming the Tooth dataset ($\approx$ 20 FPS)

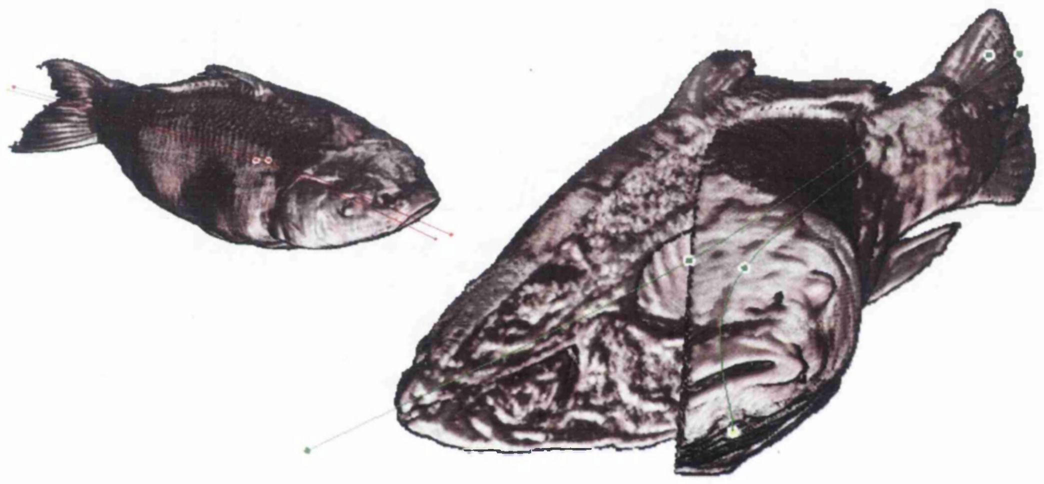


Figure 6.12: A split of the CT Carp dataset. The point of the split is chosen as a by-product of the mapping field process; voxels will attach themselves to their closest wire offset ($\approx$ 8FPS)

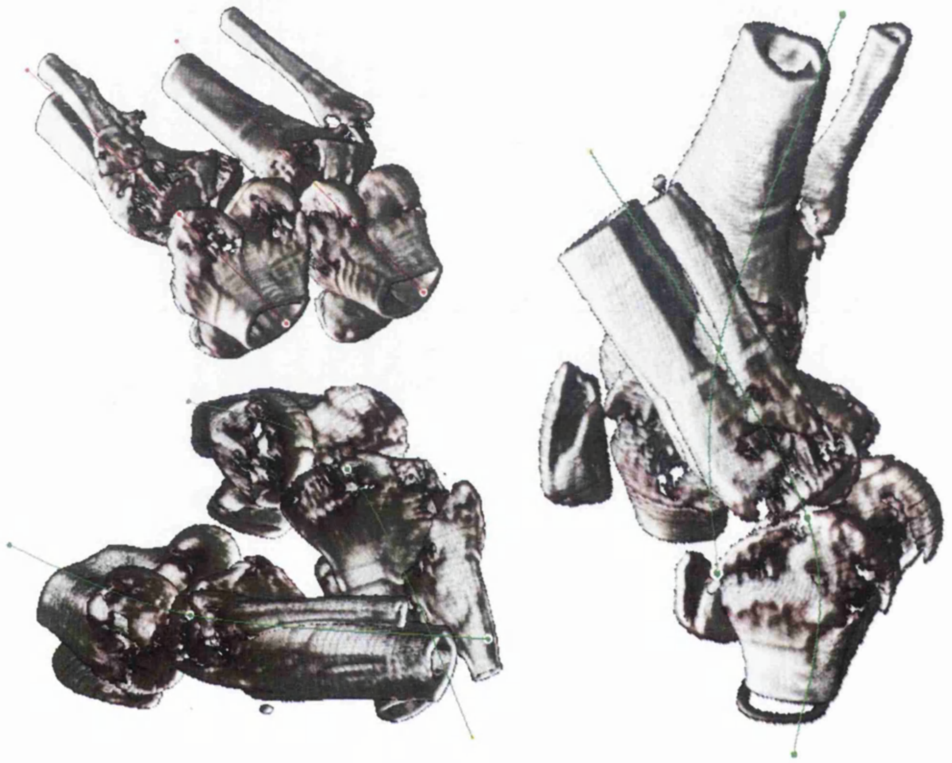


Figure 6.13: Deforming the CT Knee dataset. Segmentation was not required in this case as the legs were sufficiently separated in the dataset ($\approx 14FPS$).

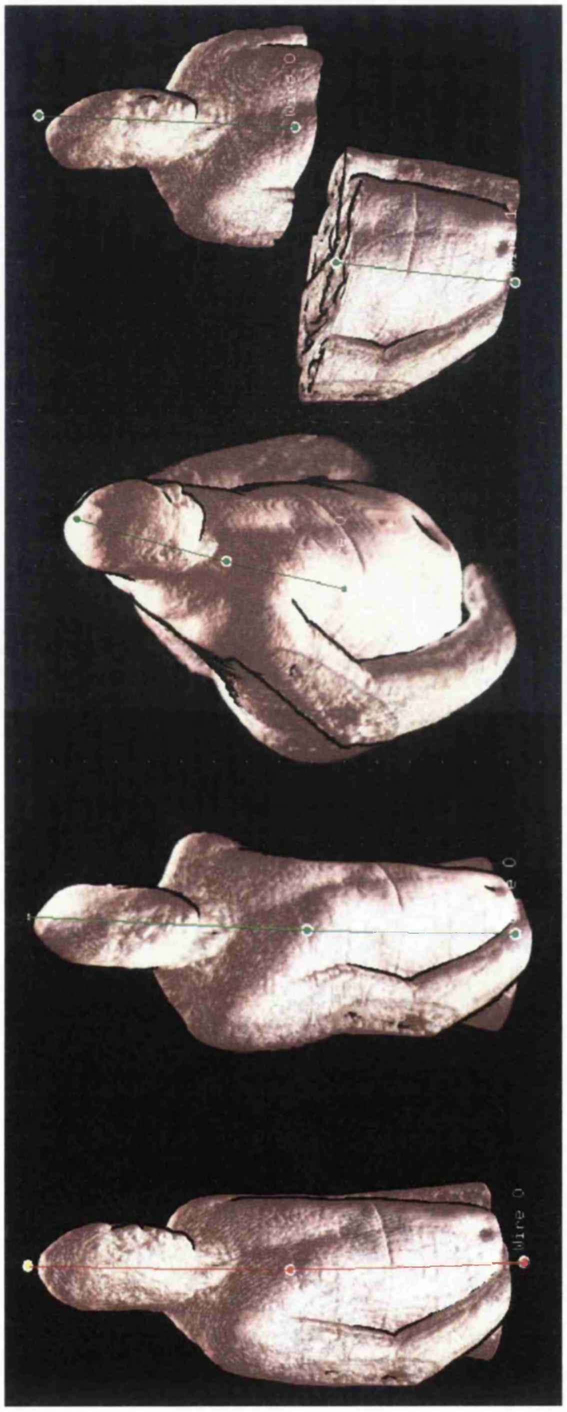


Figure 6.14: Manipulation of the Visible Human dataset

6.7 Summary

This chapter has introduced a forward-projection rendering algorithm for the volume deformation methodology *Volume Wires*. The rendering algorithm offers a high amount of interactivity, and enables the user to deform the volume object interactively. The only intermediate data that is created for the mapping process is a mapping field created from the set of all object wires that facilitates the association of voxels to their nearest wire offsets.

The work in this chapter has focused almost entirely on the technical aspects of forward-projecting volumetric datasets with interactive deformations. As such, only limited attention has been given to the visual quality of the rendering algorithm based on splatting since many such rendering algorithms already exist. A review of the most popular forward-projection algorithms, with a particular emphasis on GPU implementations, is given in Section 4.4.

Special attention has been paid however to solving the issue of cracks in image space caused where the density of samples in world space is not great enough. For this problem, a solution has been developed that is specialised to the Volume Wires methodology, taking advantage of the stretching of samples along the image-space trajectory of the wire to close these gaps in image-space. This solution effectively closes gaps introduced both by the stretching and bending of the wire.

Parts of the work in this chapter have been published in the Proceedings of the Fourth International Conference Medical Information Visualisation – BioMedical Visualisation (MediViz 2007) [WJ07].

Chapter 7

A Complete Volume Deformation Tool

Contents

Previous chapters have established the Volume Wires volume deformation methodology, algorithms, and a forward-projection rendering algorithm for its mapping function. This chapter introduces a complete tool for volume deformation utilising the Volume Wires methodology. Its approach is substantially different from the rendering algorithms given in the previous chapters in that it provides a completely real-time raycasting approach without the aid of a mapping field, and offers built-in segmentation functionality to provide the user with a means of adding useful semantic information to the volume dataset.

7.1 Introduction

In the previous chapters of this thesis, methods for evaluating the deformation specified in the Volume Wires methodology were discussed, all of which utilised the mapping field encoding of the deformation in some way. The manner in which the mapping field encoding was used depended on the direction of projection, and consequently, the mapping direction:

- The backward-projection renderer used the mapping field to evaluate Φ^{-1} while rendering to obtain the nearest world wire point;

- The forward-projection renderer used the mapping field in a pre-computation stage to link each voxel with its nearest object wire offset. Φ was then evaluated at render time based on the linked offsets.

The forward-projection renderer has the advantage that the bulk of the computation (the mapping field distance propagation) is precomputed when the user defines the object wires; the user is free to manipulate the dataset at render time and can view a constantly updated rendering of the deformation because the deformation function Φ simply requires the (γ, t) values encoded into the voxel. This renderer however has the disadvantage that the only compositing taking place is inside the ϵ boundary due to the limitations of the GPU.

Conversely, the backward-projection renderer given in Section 5.7 gives a very high rendering quality, but at the expense of user interaction as the mapping field must be recomputed every time a world wire is modified. A pitfall of rendering the deformation directly from the mapping field additionally is that the complete method is not easily parrallelisable – for this to be the case, the generation of the mapping field would need to be split into chunks and sent to separate processes for computation. Current distance transform techniques are inherently serial, although there has been some limited research conducted into separable distance transforms [Rag93].

This chapter therefore introduces a complete tool for volume deformation that provides the best of both rendering algorithms. The rendering algorithm used is a high-quality raycasting algorithm with 32-bit floating-point compositing operations that computes the backward-mapping of the deformation in real-time without the aid of any intermediate data (the mapping field). In addition to the real-time rendering, the tool also allows the user to associate interactively segmented subvolumes with each wire, allowing for greater flexibility.

The tool has been implemented primarily on GNU/Linux x86 (Debian) using OpenGL and NVIDIA's Cg toolkit. Our choice of GUI framework is Trolltech's QT framework, which provides an object oriented approach to the placement of user interface controls and user input. The toolkit is also cross-platform, and since the rendering algorithm is implemented in OpenGL, provides a relatively simple means to port the tool to other platforms such as Windows and Mac OS X (where capable GPUs are available).

7.2 Related Work on Visibility Sorting

The rendering algorithm presented in this chapter uses a visibility-sorting technique. The ultimate goal of visibility sorting is to take a list of primitives and produce a sorted list based on the inferred depth of each object. The inherent complexity of such a task however prohibits simple sorting algorithms; it is not always true that two objects are comparable in the case that they do not overlap in image-space, and additionally some objects may intersect / overlap and form cycles.

Max *et al.* [MHC90] give a software-based algorithm for rasterising and compositing polyhedra obtained from volume data by analytical computation of the ray integral at each fragment, and depth sorting of the polyhedra using a customised priority graphic algorithm. Stein *et al.* [SBM94] present a hardware-based approach to tetrahedral cell projection based

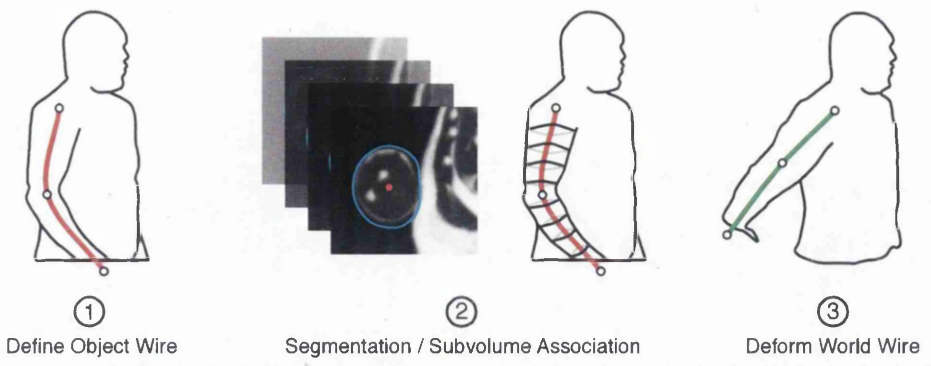


Figure 7.1: Step by Step guide for the Skeletal Volume Deformer

on the Shirley-Tuchman algorithm using compositing and texture mapping. The authors also describe an n^2 complexity algorithm for sorting the primitives. Weiler *et al.* [WKE02] take a different approach and provide a solution for current consumer graphics hardware. Their approach uses a rasterisation-based ray setup technique (see Section 4.2.1 for further details) to compute the ray inside each cell. The method however is unable to deal with optical models requiring blending; the sorting of tetrahedral meshes is not considered.

Sorting objects in terms of their depth is an atypical sorting problem as the relationship between two objects is not always defined; two objects not interacting in image-space do not have a relationship and therefore cannot be compared. Naga *et al.* give the efficient Vis-Sort algorithm for sorting objects in a 3D scene [GLM04]. In the paper, the authors give an example of an efficient GPU solution that takes advantage of occlusion queries. The algorithm also detects cycles efficiently and drops out when one is discovered. Callahan *et al.* introduce the *k*-buffer [CC05] designed for unstructured volume rendering. Their overall algorithm first performs a 'rough' visibility sort of the primitives at the object level, and then utilises the *k*-buffer for further refinement at the fragment level.

7.3 Method Pipeline

Figure 7.1 gives an overview of the three-step pipeline for the tool introduced in this chapter, which is called the *Skeletal Volume Deformer*. The definition of the object wire and world wire are already familiar from previous chapters, but an intermediate stage (step 2) is now introduced to allow for greater flexibility by allowing the user to define portions of the volume to be associated with each wire.

Upon loading the Skeletal Volume Deformer, the user is presented with the option of either selecting a volume dataset to work with, or loading a previous deformation specification (that is, the state of the object and world wires, and segmentation information) from an XML file. The complete XML schema can be found in Appendix A. The XML format stores the path and filename of the volume dataset being deformed, all of the object and world wires, and additionally the vertices of each snake defined in the segmentation stage. Volume dataset transfer functions are associated as a file *<filename>.transfer* in the same directory, which is a user-configurable plaintext file containing mappings from the $[0, 1]$ range to $< R, G, B, A >$ values.

Assuming that the user has just loaded a volume dataset and associated transfer function into the system, the pipeline is as follows (referring to Figure 7.1):

1. *Define Object Wires* – the user defines the initial curve-skeletons for the volume dataset;
2. *Associate Subvolume* – for each object wire, the user defines the portion of the volume to be associated with that wire (e.g. the arm of the Visible Human);
3. *Deform World Wires* – the user is now free to modify the world wires and view the deformation of each subvolume.

The user defines Catmull-Rom object wires by clicking with the left mouse button on the volume dataset; each click adds a new control point, with the intermediate spline being drawn as the points are added. Since the clicking is performed on a 2D screen, a suitable depth value must be chosen for the control point before it is brought from view to world space. This is achieved by projecting a ray from the view plane (at the point of the click) into the screen through view space and taking regular samples along the ray. The final depth chosen is the mean depth of the first region at which nonzero-opacity voxels were discovered. For example, clicking the arm of the Visible Human in Figure 7.2 would produce a point halfway into the arm in view space. If no such data is found along the depth of the ray, the system chooses a point as close to the centre of the volume datasets as possible.

Once the user is satisfied with the current wire being added, the enter key completes the process by adding the wire to the system. Since Catmull-Rom splines are evaluated only between the second and last -but-one control points, the computation of additional 'invisible' endpoints for the spline evaluation is carried out automatically to make the interface as intuitive as possible.

The next step in the system is to define the portion of the volume that should be associated with the wire. In Figure 7.1, the initial object wire has been defined inside the right arm of the Visible Human, and the arm of the Visible Human has been chosen to be associated with the wire. This association is performed by using built-in segmentation functionality, and is discussed in detail in the next section.

Once a segmented subvolume is associated with each wire, the user is able to drag the control points of the wire around (or drag the entire wire by holding down the shift button), and obtain an interactive raycasted rendering of the deformation.

7.3.1 User Interface

Figure 7.2 gives a screenshot of the user interface for the Skeletal Volume Deformersoftware. To navigate the volume dataset as easily as possible, a trackerball-like functionality is provided in the system; clicking and dragging on the scene with the right mouse button rotates the dataset around in the viewport. In addition, the renderer provides a perspectively-projected view of the scene at all times, so a zooming functionality is provided by scrolling

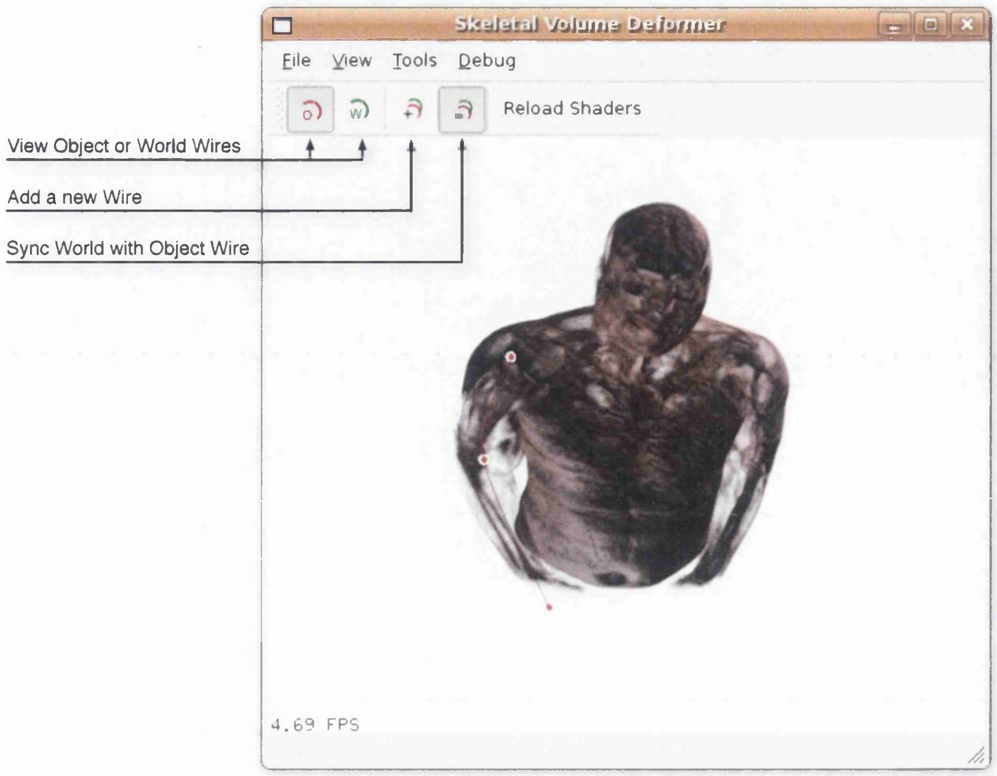


Figure 7.2: User Interface for Skeletal Volume Deformer

the mousewheel, or when not available, the up/down arrow keys.

The wires in the scene are drawn as a finite sequence of connected OpenGL lines, with a fragment program shading the lines a darker colour as the depth increases. The control points for the wires are drawn as circles (drawn using a fragment program and the point sprite OpenGL extension), with a different colour chosen for the currently selected control point.

The user is able to split a world wire into two by right-clicking on a control point and selecting the split option from the context menu that appears. In addition, the context menu offers the user with a *twist* option that applies rotation to world wire control points, thus allowing a twisting operation which is linearly interpolated between control points. A tapering operation is applied in a similar manner by applying a scaling value to each control point.

7.4 Segmentation Functionality

The segmentation functionality that allows the user to associate portions (or *subvolumes*) of the volume with each wire is integrated into the system, and the user is able to modify their subvolume specification at any stage.

The specification of each wire subvolume is achieved using a segmentation technique based upon energy-minimising Snakes [KWT88]. Since Snakes are usually 2D (though more complex polygonal 3D implementations such as Cohen's Balloons [TK95] do exist), a slice-based approach is used to gradually build up an approximation of a 3D surface (as discussed in Section 3.8.8).

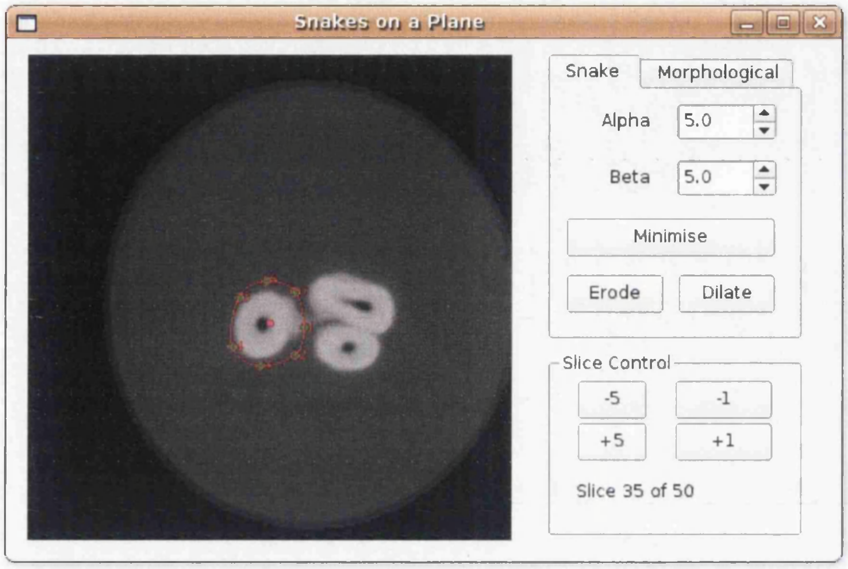


Figure 7.3: User Interface for the Built-In Segmentation Functionality

Figure 7.3 shows a screenshot of the subvolume association dialog window, which is invoked by right-clicking on a wire and selecting *'Associate subvolume...'*.

7.4.1 User Specification of Snakes

The central idea is that each snake is defined on a finite plane, and there are multiple planes defined along the trajectory of the object wire; hence the system is dubbed 'Snakes on a Plane'. The software defines a series of k planes along the trajectory of the wire, all equally spaced. Each plane's normal is equal to the wire tangent, and the precomputed and corrected wire normal (see Chapter 5, section 5.4 for details) is chosen as the 'up' vector for the plane. Associated with each plane is a 2D snapshot of the scalar volume data cut by the plane; essentially a 2D greyscale image [1]. The image is obtained quickly from the plane definition and volume data using pointer arithmetic and nearest neighbour interpolation.

Initially when the user interface is shown, the first slice image is shown to the user; that is, the image obtained from the plane positioned at offset $t = 0$ on the object wire. The vertices of the Snake are added by clicking the left mouse button on the plane image, and can be later moved with the right mouse button. The *minimise* button minimises the Snake

[1]To assist with the segmentation process, an adaptive nonlinear diffusion filter is first applied to the volume dataset

around the object of interest. The minimisation algorithm employed uses a local minimisation approach, attempting during each step to find the minimal setup of vertices for a small radius defined around each Snake vertex. The image energy functional employed is an edge attraction functional based on the gradient of the pixel as $E_{image} = -\nabla I$.

A satisfactory minimisation of the snake around the object of interest may take more than one minimisation attempt, but the user is able to fine-tune the minimisation process by altering the Snake's α and β parameters. The α and β parameters of the Snake control its continuity (the *elastic force* of the Snake) and curvature (the energy caused by bending), respectively, and can be modified by the user using the spin boxes present on the right hand side of the dialog. These parameters have a large effect on the shapes that the Snake can form; unfortunately in practice however, the modification of the parameters to fit the desired result is often rather arbitrary.

Once the user is satisfied with the snake definition, they can advance the slice index to show the next slice in the slice series. Upon selecting a new slice, the Snake defined from the previous slice is copied to the new slice, and attempts to minimise itself around the newly discovered data; the assumption behind this behaviour is that the object of interest has not changed substantially since the previous slice, and therefore a minimisation attempt will most likely be successful.

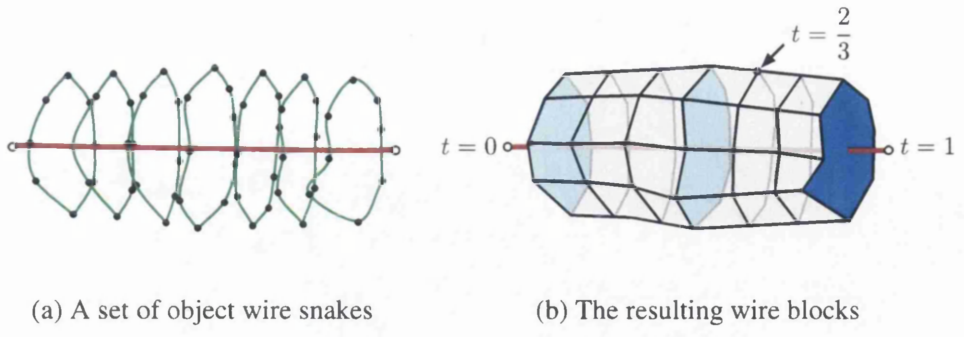
7.4.2 Additional Tools

Alongside the basic minimisation functionality, the user interface dialog presents some basic Snake manipulation tools for quickly dilating and eroding the snake vertices (moving the vertices away and towards from the the 'mean' vertex position, respectively). The tool also allows for basic manipulation of the 2D slice image used by the Snake minimisation process, which is a particularly useful operation for noisy data. The 'Morphological' tab allows for thresholding operations to be applied to the image, as well as image dilation, erosion, and closure operations. These operations often have a substantial effect on the final result where the boundary of the object of interest is not well defined.

7.4.3 Subvolume Polygonal Approximation – Wire Blocks

The end result of the segmentation step is a set of Snakes defined along the trajectory of the object wire, as shown in Figure 7.4(a). These Snakes however do not define a continuous boundary of the desired object of interest by themselves. In order to form a continuous representation of the subvolume boundary, a polygonal mesh is fitted around the Snakes, which is defined by joining the Snakes together. For implementation reasons discussed in detail later, this polygonal mesh is further decomposed into a series of *blocks*. Figure 7.4 shows two such blocks defined along the trajectory of a wire; the blue faces indicate the boundary between blocks.

This is an example of a 2D-based segmentation methodology applied to 3D, where slices of segmentation data are 'stitched' together; a review of such techniques has already been given.



(a) A set of object wire snakes (b) The resulting wire blocks

Figure 7.4: The subvolume boundaries are approximated using polyhedral blocks. The wire offset t is encoded into each slice's vertices.

Since the Snake splines defined around the object are higher-order splines, a polyhedral mesh will only approximate the final subvolume; the density of vertices in the polyhedral mesh governs the accuracy of the approximation. The polyhedron is defined as two polygonal faces (created from the first and last snake splines), connected via a triangular strip that approximates the Snake splines in between.

The polygonal structure is built by discretising each Snake spline. This is achieved by evaluating the spline at a fixed interval, giving a set of discrete points. The control points are also an acceptable 'discretisation'; though through experimentation it was discovered that this method relied too heavily on the number of control points defined by the user. The first Snake spline is discretised into a relatively sparse set of points. For each vertex on this first slice, the algorithm attempts to find the offset on the next Snake that minimises the distance between them. Once this offset is found, a vertex is created on the next slice at the minimal offset, and a temporary connection is made between the two vertices.

Ambiguities are dealt with by ensuring that each vertex is connected to only one other vertex. This continues for each discretised vertex on the first slice. Once complete for all vertices, a triangular strip is created between the two slices by joining using the connectivity information. This process continues until a complete polygonal representation of the segmented subvolume has been constructed. It should be noted that this method is sensitive to topological changes, and the splitting / merging of Snakes is unsupported.

The polyhedron is now decomposed further into blocks of fixed size k (with a possibly variable remainder sized block at the end). For each k slices connected with triangular strips, two polygonal face 'caps' are created from the associated slices to create a closed polyhedron from the k slices. These caps are represented in Figure 7.4 as blue faces. Once each block is specified, it is saved as an OpenGL display list for simple invocation in the rendering algorithm.

The rendering algorithm introduced later in the chapter requires, for a given rasterised fragment belonging to a block, that a reasonably accurate t-value can be obtained for the fragment which represents the nearest wire offset. Since each snake is associated with a plane at a particular offset along the wire, this offset can be encoded into the R colour component of each vertex on that slice. When this representation is rendered and the correct blending operations are specified in OpenGL, these values will be linearly interpolated between slices.

In addition, parts of the rendering algorithm that deal with normal computation require that the normals of each cap are correctly specified. These normals are computed by choosing three of the vertices on the cap, creating two vectors from these vertices, and then taking the cross product. Care is taken to ensure that the normal is correctly pointing 'out' of the block.

7.5 Deformation Rendering Algorithm

The rendering algorithm employed in this chapter is substantially different from the algorithms presented in the previous two chapters for two main reasons. Firstly, the tool introduced in this chapter allows the user to define segmented portions of the volume to be associated with each wire; this needs to be taken into account during rendering to ensure that the correct portions of the volume are deformed. Secondly, the goal of the tool is to give the user a complete, interactive raycasted view of the deformation process, without any intermediate delays. Thus, a real-time raycasting rendering algorithm is required that evaluates the Volume Wires backward-mapping function on-the-fly.

In this section, a GPU-based rendering algorithm is introduced that implements these requirements. The algorithm exploits the segmentation information defined by the user to obtain optimisations for the mapping process, which is discussed in detail in coming sections. The raycasting/mapping operations introduced require ShaderModel 3.0-capable hardware, as extensive use is made of looping and dynamic branching instructions. A ShaderModel 2.0 implementation of all shaders has been evaluated to be possible, but would result in less efficiency. In addition, several ShaderModel 3.0 optimisations make its implementation much more favourable.

Framebuffer Objects (FBOs) are used in the implementation for the compositing operations, and also to store intermediate ray entry and termination points. The compositing buffer stores the composited colour in the $< R, B, G >$ components and the ray's current opacity in the α component. The ray entry / termination point buffers are alternated using a ping-pong technique (swapping the roles of the buffers for each pass), as reading and writing to a single buffer is undefined and therefore discouraged. The internal format used for the Framebuffer Objects is a 32-bit floating-point internal format that provides a high precision of compositing.

7.5.1 Algorithm Overview

The rendering algorithm presented in this section is a hybrid surface and volume-based algorithm which makes extensive use of the polyhedra generated during the segmentation step to generate the entry and exit points for the rays to be fired through the segmented portions of the volume. This rasterisation-based ray setup method was first devised by Kruger and Westermann [KW03] for standard volume rendering by rasterising the volume dataset

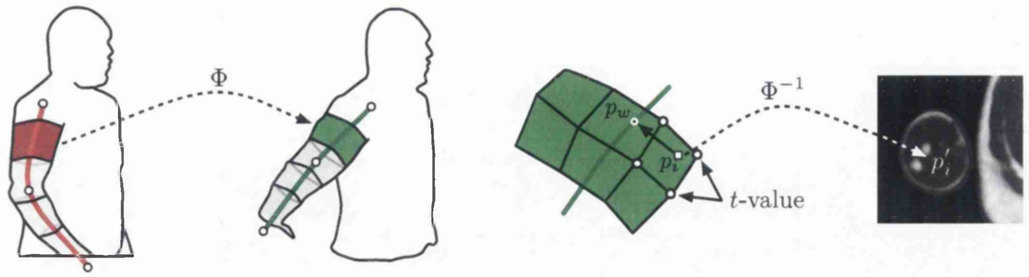


Figure 7.5: Rendering algorithm overview: The subvolume meshes are forward-mapped from object space to world space; a GPU raycasting algorithm internally raycasts these meshes.

boundary and saving the framebuffers as the ray entry / exit points (discussed in detail in Section 4.2.1 of this thesis).

Referring to Figure 7.5, an outline of our rendering algorithm is given below.

Forward-map wire subvolumes The subvolumes defined on the object wires (red in Figure 7.5) are forward-mapped into world space using the forward-mapping function Φ

Raycast subvolumes The subvolumes are rasterised **in depth order** and internally raycasted by evaluating Φ^{-1} to backward-map each ray sample point into the original volume dataset.

The remaining parts of this section are structured as follows. First in Section 7.5.2, a definition of what comprises a scene in the system is given based on the segmentation information. Sections 7.5.3 and 7.5.4 next detail the process of forward-mapping the wire subvolumes into world space and the depth-sorting procedures used on them. Finally, Sections 7.5.5 and 7.5.6 detail in the main portions of the rendering algorithm.

7.5.2 Scene Definition

To recap, each polyhedral representation of the subvolumes is decomposed into a series of *blocks*. In a traditional volume renderer, the ray entry and exit points are typically defined by the dataset boundaries. If an octree is used, then the process is changed such that the applicable octree blocks generate the entry and exit points at each stage depending on their contents. For rendering algorithms that deal with unreconstructed deformed data, the generation of ray entry and exit points is not always straightforward as it is not immediately obvious where the boundary of the deformation lies.

Due to the segmentation step involved in this system, the new positions of the deformed objects represented by each wire can be approximated by forward-mapping (using Φ) the applicable wire blocks from object space (where they were defined segmenting the volume data) into world space (where the deformed model and scene exist). Once these blocks are in world space, the only task that remains is to discover the volume data that should lie within the block interior by projecting rays through the block, aligned with the viewer. For each ray sample point inside the block, the sample point is backward-mapped using Φ^{-1} to discover the true sample point in object space (which will always lie within the boundary of the block when in object space).

The above process effectively takes care of the segmented portions of the volume, but additional consideration is required for other regions also:

- First, regions of the dataset which we refer to as *static*; that is, regions that have not been encapsulated by a wire block during the segmentation step and therefore remain static in the dataset. Such regions must be raycasted 'as-is' without evaluating Φ^{-1}, and must be able to integrate with the deformed volume data in the scene;
- Second, regions of the dataset which have been segmented by the user and no longer contain the segmented data due to the deformation, which we refer to as *abandoned*; for example, in Figure 7.1 (step 3), the region where the visible human's arm *used* to be before it was moved away by deforming the world wire – such a region is only empty in world space, but not object space.

Let O be the set of all points within an object block, W the set of all points within a world block, and D the set of all points within the volume dataset. A more formal definition of the *static* regions can be defined as $D - (O \cup W)$; that is, the entire volume dataset, minus the object and world blocks. The *abandoned* regions can be defined as $O - W$; that is, all object blocks regions except where a world block subvolume intersects it.

7.5.3 Forward-Mapping the Wire Blocks

The first step of the rendering algorithm must forward-map each wire block into world space based on the current deformation specified by the state of the object and world wires. This process simply involves invoking Φ on each block vertex and creating a new block in world space using the new vertex positions. This can be achieved efficiently as the nearest wire offsets for each snake vertex are explicitly set by the segmentation step and are therefore available to Φ without any distance computations. The mapping performed on the vertices is the full Volume Wires mapping, including the rotation and scaling effects, as specified by the derivation in Section 5.4.2.

Figure 7.6(a) shows the wire blocks rendered as wireframe meshes. The object blocks are shown along the sides of the body, and were defined during the segmentation step. In this example, the user has stretched both arms out to the side, which in turn repositions the associated world blocks as they are mapped into world space. Figure 7.6(b) shows the resulting render from the scene using the rendering algorithm discussed in this section.

The forward-mapping of wire blocks into world space could easily be performed by the GPU. However, the next stage of the algorithm requires a CPU depth-sort of all blocks such that the block can be processed on the CPU side in depth order, and therefore, performing Φ on each block vertex would be insufficient. A possible speedup could be obtained by writing the new vertices to the framebuffer and then binding the resulting buffer as a the new vertex buffer representing that particular block. It is unclear however how much of a speed improvement this would achieve over simply performing the mapping on the CPU,

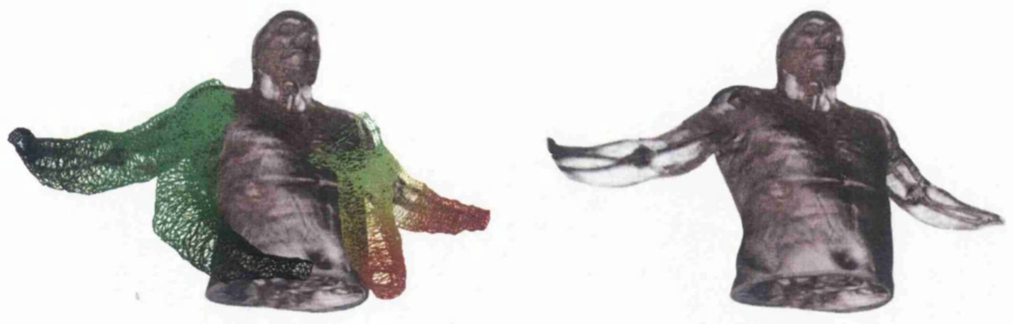

(a) With blocks shown (b) Final render

Figure 7.6: The object and world meshes defined for the Visible Human.

since the number of vertices involved is small enough – typically less than a thousand. The decision was made therefore when implementing the system to perform the forward-mapping of blocks on the CPU rather than the GPU.

7.5.4 Depth-Sorting

The second step of the rendering algorithm involves a depth-sort of all wire blocks defined in the scene. The input to the sorting algorithm is a list of all blocks; there is no distinction made between object and world blocks. The sorting algorithm used in the system is based on the Vis-Sort algorithm [GLM04], which provides a near-linear time sorting algorithm for nonintersecting primitives. In the case of intersections between blocks therefore, it is clear that a decision procedure must be able to choose one block over the other.

The visibility sorting procedure makes extensive use of OpenGL occlusion queries to determine whether a particular object is in front of another. Occlusion queries are a useful mechanism for asking the hardware how many fragments made it through the pipeline based on the fragment tests; in this case, the depth test is the most relevant. To test whether object a is fully in front of another, object a is drawn with no depth operations and the number of passed fragments counted. Next, object b is drawn, followed by object a again, this time with depth testing and writes enabled. If the number of fragments passing is the same as previously measured, then it follows that object a is fully in front. Face culling has an important effect on the results also - it is important to distinguish the results of comparing just the front faces of an object with all faces of the object. For the depth-sorting procedure employed in this system, face culling is enabled to remove back-facing polygons.

For the purpose of this system, the Vis-Sort algorithm given in [GLM04] must be modified to account for the case of overlapping blocks, since the Vis-Sort algorithm assumes that such overlaps do not exist. Therefore, the comparison function for two blocks cannot simply be a comparison between the numbers returned by the occlusion queries. The depth-sorting of wire blocks has an important effect on the final image, as some world wire blocks (and consequently, the associated deformed volume data) will take precedence over other intersecting world blocks in the final list. In the event that two world blocks intersect, a decision has to be made as to which block takes the front position. The following rules are therefore defined for intersections:

- *world* and *object* block – as will be discussed, the exact order between the two is unimportant, but the world block is chosen as the winner
- *object* and *object* block – the block with the lowest mean image-space z-coordinate
- *world* and *world* block – the block with the lowest mean image-space z-coordinate

There is a possibility of popping artefacts occurring due to the third rule defined above – although this can be minimised by the possibility of implementing a CVG scheme (discussed later in the Chapter).

7.5.5 Stage One: Initial Raycasting Probe

Once the wire blocks have been created and sorted by depth, the main rendering algorithm must take each block in turn and use rasterisation of the block geometry to advance rays through world space; much in the same manner as the standard rasterisation-based raycasting techniques. The exact ordering of rasterisation and the blocks chosen for rasterisation at each stage determines the starting and ending points for each ray.

Figure 7.5 shows a cross section of all wire blocks defined in the scene, including the dataset boundary and the static data within the dataset. *Static* data is defined in this context as data which has not been segmented by the user for potential deformation – e.g. the torso of the Visible Human in Figure 7.1. Such data must be rendered without evaluating Φ^{-1} and included in the compositing steps. In Figure 7.5, the static data is everything except the right arm of the human model.

The rendering algorithm itself is broken into two stages – namely the *probe* stage (discussed in this section), and the *block rasterisation stage*, discussed in Section 7.5.6. The compositing buffer is initialised to all zeroes.

Ray Instantiation

The first step of the probe stage is to instantiate rays for the scene and fire these rays until they hit either a wire block or one of the back faces of the dataset. Referring to Figure 7.5, a ray is instantiated for a given pixel in the framebuffer if the pixel is hit by either a rasterised object or world block, or the dataset boundary. Instantiating rays based purely on the latter case would often result in an incorrect rendering, as it is possible that the user will have deformed part of the volume object outside of the dataset boundaries.

The procedures for instantiating the initial ray entry / termination points are given below. The entry points are written to one Framebuffer Object, and the termination points to another.

Entry points The depth buffer is first cleared to 1 and the depth test is set to $\leq$. The front face of the volume bounding box is now rasterised, followed by all wire blocks (object & world).

Termination points The back faces of the volume dataset boundary are rasterised, followed by the front faces of all blocks.

The ray entry and termination points are output by a fragment program running on the GPU that converts the world-space coordinates of the fragment (given to the fragment shader by the vertex shader, encoded into a texture coordinate) to the output $< R, G, B >$ triple for the framebuffer. In addition, the length of the ray is output into the α component of the entry point Framebuffer Object for later determination of ray termination based on the ray's progress through the block.

Firing the Rays

Once the initial ray entry and termination points are computed, the rays are fired from the entry to the termination points. Some rays will not be fired – for example, rays created for world blocks outside of the volume dataset boundary. Such rays will have length 0 and can be immediately terminated. The useful work in this probe stage therefore is raycasting any static data that is encountered in the dataset before any blocks (either object or world) or the back dataset boundary is hit.

Algorithm 4 gives this raycasting procedure.

Algorithm 4 CPU Invoking raycasting shader

```
 1: CPURayStep = 0;
 2: repeat
 3:     {Run termination shader first}
 4:     Bind Termination Shader
 5:     Rasterise back faces of $Block_j$
 6:     DepthFunc : =
 7:     {Next, run the raycasting shader}
 8:     Bind Raycaster Shader
 9:     Rasterise $Block_j$
10:     {Get number of fragments passing depth test}
11:     Passed = OcclusionQuery();
12:     {Advance the ray section index}
13:     CPURayStep += STEP_SIZE;
14: until Number of fragments passing depth test = 0
```

This loop iterates until there are no more rays to process. The algorithm takes advantage of early-Z termination in a similar fashion to the GPU raycasting algorithm introduced in Section 5.7.

The termination shader simply checks the ray's progress (based on the $CPURayStep$ parameter passed into it) against the ray's length saved into the α component of the entry point buffer by the ray instantiation stage. If the number of unit steps are greater than the ray length, then the shader outputs a fragment of depth 0 to cause the ray to fail the depth test in future attempts. This termination shader additionally acts as an adaptive ray termination system (first discussed by Levoy [Lev90a]), halting the ray when its opacity reaches a predefined threshold (set to 0.95).

The raycasting compositing operations are performed on the CPU in steps, with variable $STEP\_SIZE$ determining the number of ray steps performed each time the shader is invoked in the CPU loop. The GPU raycasting algorithm itself is based on the implementation given in Section 4.2.1. Note that the depth test is set to $=$ before casting rays to ensure that only one ray per pixel is fired for the case where more than one front face is rasterised to the same pixel (for non-convex meshes).

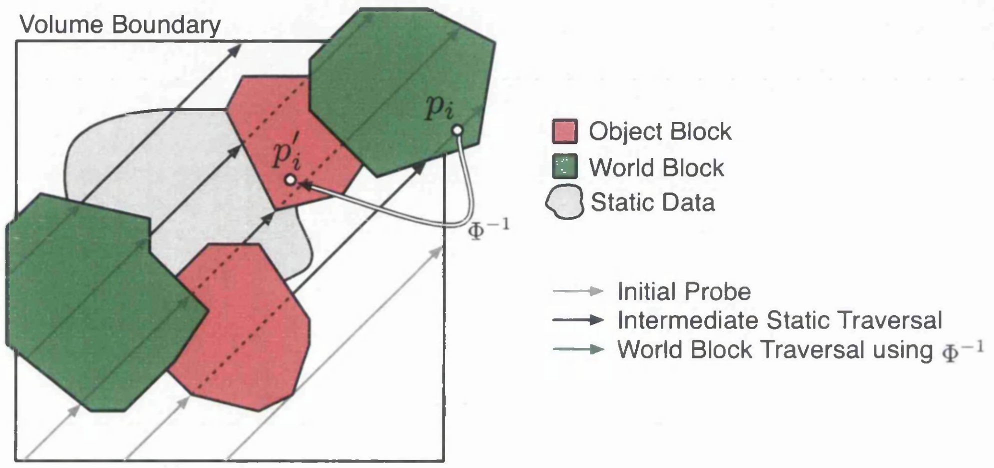


Figure 7.7: Rays in the scene negotiating the static volume data (grey), object blocks (red) and world blocks (green). Inside world blocks, Φ^{-1} is applied to each ray sample point p_i obtain the new sample point p'_i from its associated object subvolume.

The initialisation of the next rays to be fired through the scene is made by the wire block rasterisation stage, discussed in the next section.

7.5.6 Stage Two: Wire Block Rasterisation

The aim of this stage is to render the deformed subvolumes by processing each wire block in turn contained in the depth-sorted list. This stage therefore requires consideration of not only the wire blocks attached to world wires (where the ray sample points must be backward mapped using Φ^{-1}), but also of the original object wire blocks defined during the segmentation step.

The algorithm is based around a main loop that iterates through each block contained in the depth-sorted wire block list. In order to generate rays where appropriate, the blocks are rasterised to the framebuffer and the world coordinates written as the $< R, G, B >$ values, similar to the previous ray instantiation stage. The handling of the block depends on its type (whether it is an object or a world block), and the procedure for each block type is given below.

Object Blocks define the static, segmented subvolumes. Each ray must skip over these, as it is their world representation that must be rendered. More correctly, the rays must skip over the difference of the object blocks and the world blocks; since any intersection of a world block implies that there exists freshly deformed data.

World Blocks define the dynamic subvolumes deformed around the world wires. The renderer must send rays through these blocks and perform the backward-mapping operation Φ^{-1} on each sample point p_i, to discover the sample point p'_i in the volume dataset. This backward-mapping process is discussed in detail in Section 7.5.7.

Algorithm 5 gives the block rasterisation procedure employed in the system. The framebuffer objects used for storing the ray entry and termination points are referred to as Ray_{entry} and Ray_{term} respectively. The code for swapping the roles of these buffers is omitted for clarity.

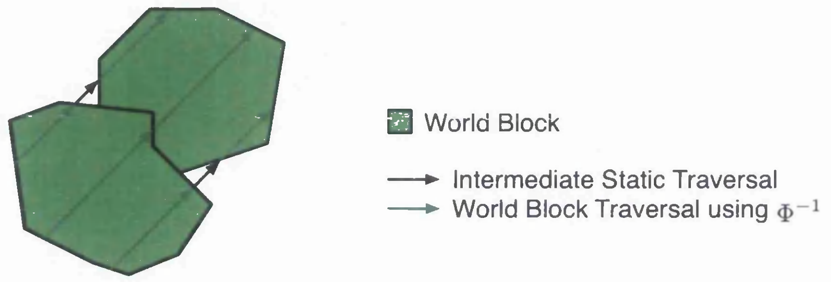

World Block
Intermediate Static Traversal
World Block Traversal using Φ^{-1}

Figure 7.8: Intersecting World Blocks

The first half of the algorithm (lines 2 to 18) deals with the backward-mapping raycasting inside world blocks, shown as solid green lines in Figure 7.7. Each ray always terminates at the back face of the world block. It is possible however that world blocks will intersect each other; in this case, there will be some rays that will begin their progress at a lower depth (closer to the viewer) than where they terminated at the back face of the previously rasterised world block. If a terminated ray from the previous ray block has a greater depth than the current ray, then the ray for the current block will need to begin at the previously-terminated depth. This is illustrated in Figure 7.8.

To correctly generate ray entry points for a world block therefore, the front face of the current world block is first rasterised. Next, the depth test is set to $>$, and the back faces of all obscuring world blocks are rendered to capture any possible ray termination points that have the greater depth. The termination points for a world block are simply at the back faces of the block.

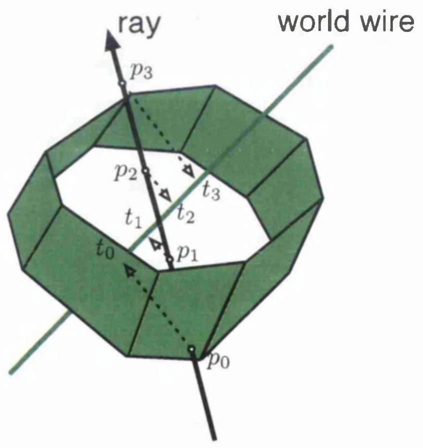
The second half (lines 20 to 35) deals with any static data between blocks, shown as solid black lines. The ray starting points for rendering the static data must be the back faces of $Block_i$ (whether it is an object or a world block), except in the case that an object block is intersecting it. If this occurs, then the ray will need to begin at the back face of the object block as the data inside the object block must not be rendered. The termination points for the rays are generated from the front faces of any object or world block behind $Block_i$.

7.5.7 GPU Backward-Mapping

The largest block of fragment shader code, the GPU backward-mapping raycaster, is invoked when a world block is rasterised (Algorithm 5, line 17). The purpose of the GPU backward-mapping raycaster is to fire rays through the world block, establishing new sample points in object space using Φ^{-1} and compositing the results in the compositing framebuffer object. Line 17 of Algorithm 5 is broken down into several steps which invoke this shader, and they are similar to those shown in Algorithm 4 for the raycasting of static data in the ray instantiation stage.

Refining the t-value

The key piece of information that is required for the backward mapping of a Volume Wires deformation is the nearest wire offset t given a point in world space. In previous implementations (such as that given in Section 5.7), this information was obtained from the *mapping field*, which is simply a distance field of the wires. However, it is clear that generating a new mapping field each time the user deforms a world wire would result in an unacceptable performance penalty in terms of the generation time, and the time to upload the mapping field to the GPU.



$$t_0 = \text{refine}(\text{fromFace}(p_0), p_0)$$
$$t_1 = \text{refine}(t_0, p_1)$$
$$t_2 = \text{refine}(t_1, p_2)$$
$$t_3 = \text{refine}(t_2, p_3)$$

Figure 7.9: Obtaining the t-values for the initial sample point on the face and subsequent sample points.

Since the wire offset t is encoded into the red colour component of each wire block vertex (corresponding with the slice aligned with a particular offset t along the wire), a linear approximation to the nearest wire point (at the ray entry point) is found by hardware interpolation across the polyhedron face. Figure 7.9 shows a ray entering a face belonging to a block. For the initial sample point p_0, the approximate nearest t-value is discovered from the interpolated red component. This approximation gives a good starting value since the value is interpolated linearly along the approximate trajectory of the wire.

From this, a more accurate value can be derived by a refinement procedure. A function `refine(t,p)` is defined that searches the points on the wire around t for a more accurate t-value for point p. This function can be implemented by defining a fixed-sized window around the t-value and finding the minimally-distant t-value for point p inside this window. The function is implemented in Cg using a loop, and the size of the window is set as a predefined even integer constant. Once this initial t-value has been discovered for a given ray entry point, it can be continually refined with each new sample point along the ray by sending the previously refined t-value into `refine` along with the current sample point. This gives an effective mechanism for continually refining the nearest wire point in small steps. The size of the window is derived empirically through observation of rendering quality on a variety of datasets, and the most suitable constant was found to be around 14.

Performing Φ^{-1} and Compositing

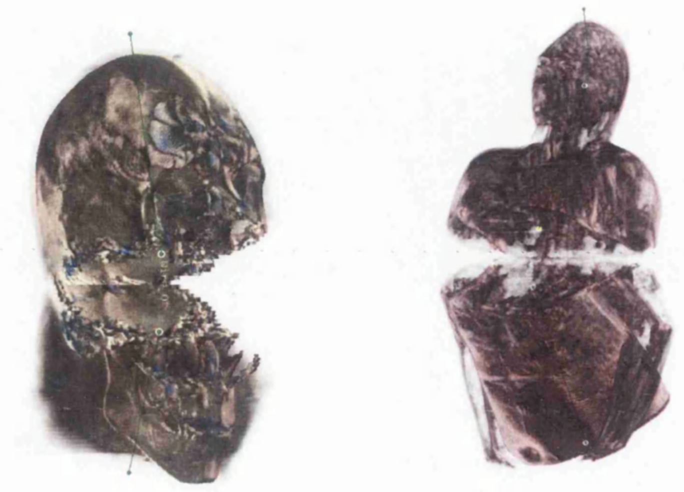
For each ray sample point p_i, the refined t-value is used to look up the associated world wire point p_w via a texture lookup on the wire texture using the t-value as the x-offset. Once this world wire point has been discovered, the associated object wire point p_o is obtained using the same offset. The other variables required for input to the mapping equation are computed using texture lookups on the wire texture, wire normal texture, and wire effects texture, in the same manner as described previously for the forward-projection system; described in Section 6.4.1.

Once the required mapping variables are available, the backward-mapping operation Φ^{-1} is computed to discover the new sample point in object space. This point is sampled using a texture lookup in the volume dataset 3D texture, and the colour/opacity values are derived from the transfer function specified by the user. These values are now used in a compositing equation to blend them with the current colour/opacity values, which were initially obtained from the compositing framebuffer object. The current sample point on the ray is now advanced, and the t-value is further refined for the next sample by using the previous t-value as an approximation.

The raycasting scheme discussed in this section operates in slices, stepping a constant number of steps along the ray before writing the results to the compositing buffer and exiting. This behaviour was first devised for GPU raycasters to allow for adaptive ray termination and to more effectively load balance the GPU's available shading units. The behaviour is especially important for the backward-mapping raycaster introduced in this section, as it additionally avoids the possibility of hitting the instruction limit of the fragment shader (2^{16} for ShaderModel 3.0).

It is clear that interpolated approximation of the t-value will only be accurate when the ray begins its journey through the block. When the block is re-rasterised for the second and subsequent passes, the interpolated value across the face will be only reasonably accurate for the intersection of the ray and the face. To solve this problem, the current t-value at the end of the previous pass is packed into the alpha component of the compositing buffer. This is achieved by dividing the compositing buffer into two 16-bit floats: the ray opacity value and the refined t-value. The packing of floats in this manner is supported by the FP40 fragment profile with a dedicated instruction. By packing values in this manner, care must be taken to unpack the ray opacity correctly in subsequent operations, and in the ray termination shader.

7.6 Discontinuities in World Space



(a) Artefacts noticeable (b) Higher resolution mapping field

Figure 7.10: Aliasing artefacts caused by encoding discontinuities into a single mapping field. Encoding the split into a higher resolution mapping field alleviates the aliasing slightly.

The Volume Wires methodology allows for the splitting of wires into two to produce an explicit discontinuity, and consideration must be given to how the wire split affects the blocks associated with the wire. Consideration must additionally be given to the computation of normals for lighting calculations in the areas where a discontinuity has been created; both in the case of an explicit split of the wire, and the case where a discontinuity has been created by moving a segmented subvolume away from a static portion of the dataset.

The advantage of using the segmentation information for splitting is that an explicit, continuous split boundary has been created; this boundary can be used during the rendering process to decide whether a ray sample point is inside or outside the data. Figure 7.10 shows an attempt to encode a split of the Visible Human's head into a mapping field and rendered using the mapping field-based raycasting renderer introduced in Chapter 5. The blocky aliasing artefacts are noticeable in Figure 7.10(a) in the areas near the split, and are caused by the signal reconstruction (trilinear interpolation) taking place in object space, which does not have the discontinuity and thus for the object interior cannot give a smoothly blended object-to-empty-space value. This problem can however be alleviated slightly by using a higher-resolution mapping field, as shown in Figure 7.10(b), where the blocky artefacts are not as noticeable.

This section looks at the splitting functionality in the Skeletal Volume Deformer, and additionally looks at the problem of computing correct normals near the split.

7.6.1 Wire Splitting

Many interesting effects can be achieved by splitting a wire in half to reveal the interior of the object defined along the wire trajectory. This section reviews the implementation details and caveats of splitting wires.

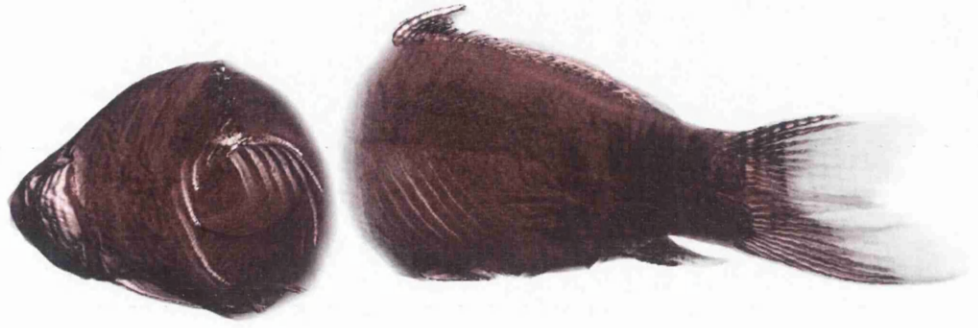


Figure 7.11: Splitting the CT Carp in half, revealing the interior.

A wire split is achieved by choosing a control point, and using the context menu on that control point to select the 'cut wire here' option. If control point at index i is chosen as the cut point and the wire has n control points, then *in the general case* this operation involves creating two new wires, and copying $p_0, \ldots, p_i$ to the first and $p_i, \ldots, p_{n-1}$ to the second. We can also find the approximate t-value of the control point in question.

It is important that continuity at the point of the split is maintained, particularly for the object wires as they maintain the segmentation information. There should be no change in the mapped positions of the object blocks before and after a split.

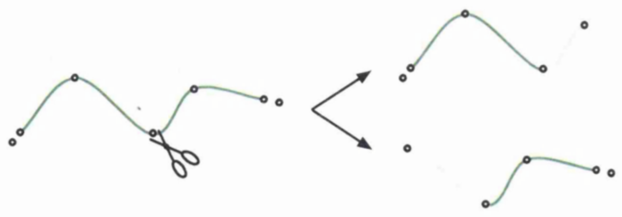


Figure 7.12: Splitting a Catmull-Rom Wire

Since the wires used in this chapter are Catmull-Rom splines, maintaining continuity in the object wires simply involves matching the 'ending' control points with the correct points on the other wire (see Figure 7.12).

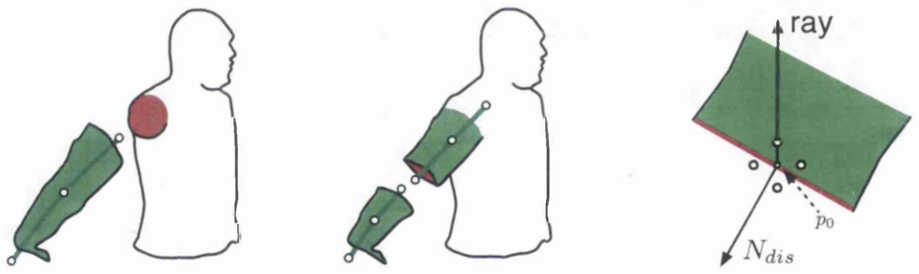
When a wire is split, the snake slices defined in the segmentation stage must be distributed correctly to each new wire. Each snake slice before the split is copied to the first new wire, and then all others are copied to the second wire. The wire block list is now rebuilt for the new wires.

7.6.2 Normals at Discontinuities

Normals in the interior of the blocks are sampled in object space and transformed with Φ_n to bring the normal into world space. The Volume Wires mapping fortunately provides a mapping that can offer such a straightforward mapping from one space to another; with many other systems that rely on analytical approaches, the Jacobian of the mapping must be computed to forward-map the normal. The alternative approach is to perform the normal calculations in world space by performing Φ^{-1} on each central difference sampling point; something to avoid when the cost of Φ^{-1} is high or there are time / calculation constraints such as those that exist in the fragment shader.

Forward-mapping the object-space normals gives accurate normals for the usual cases in the system. However, it can lead to inaccurate normals when there exists a discontinuity in world space that does not exist in object space. As illustrated in Figure 7.13, such a discontinuity can be introduced by two means:

- *Implicit* – The user has revealed a static portion of the volume that is cut by a segmented portion; or
- *Explicit* – The user has split the dataset using the wire splitting functionality.



(a) Implicit Split (b) Explicit Split (c) 2D analogy

Figure 7.13: Computing normals at discontinuities requires special handling.

The former case is illustrated in Figure 7.13(a) and occurs in situations where the user reveals a static portion of the dataset by moving segmented portions out of the way. In this example, the user has deformed the arm of the Visible Human and has then translated the entire wire away, moving the segmented arm away from the body. The split area revealed cutting through the shoulder is an implicit split in world space. The latter case is illustrated in Figure 7.13(b) and occurs when the user invokes the wire splitting functionality of the system. In this example, an explicit split is created halfway along the arm. Note that the outward-pointing faces belonging to the innermost wire blocks both mark the point of the split if they were backward-mapped into object space.

Figure 7.13(c) demonstrates an attempt to compute a normal near a discontinuity; the green area is interpreted as the volume data, the white area as the empty space *created by* the discontinuity, and the red as the split boundary. Around the main sample point, a set of points for central differencing are created to be backward-mapped using Φ^{-1}. It is clear that two of these sample points fall into empty space – space that was created by the discontinuity. Since these two points would be backward-mapped into object space and the discontinuity does not exist there, an incorrect normal is ultimately computed.

The problem of recovering correct normals at discontinuities has been discussed in detail by Weiskopf [WEE03] where a GPU-based volume clipping algorithm is implemented. The author discusses methods for approximating normals at the boundary of the clip, even for arbitrary volumetric clip templates; achieved by defining the discontinuous 'surface' to have a finite thickness. Inside this transitional layer, the normal is blended from the normal in the volume (below the surface), N_{vol}, to the normal of the discontinuity-causing surface N_{dis}:

$$N = \omega N_{dis} + (1 - \omega) N_{vol} \tag{7.1}$$

where ω is a blending factor in $[0, 1]$; higher values give a normal that is biased towards the discontinuous surface normal. In a sense, the normal is being interpolated. This approach is also used by Correa *et al.* [CSC03] for a GPU-based volume displacement mapping system that allows for cuts to be performed.

Identifying Discontinuous Boundaries

Before any normal correction can take place, the fragment shader must be able to ascertain whether the ray point it is currently dealing with is on a discontinuous boundary.

In the explicit case, the discontinuity can be flagged on the mesh itself and picked up in the fragment shader. When a wire is split into two, a flag is set with the 'discontinuous' faces to inform the fragment shader that they are discontinuous, which is achieved by setting the unused green colour component of the face to 1. When a ray initially begins from a face (where the $CPUStep$ variable is 0) and the 'discontinuous' flag is set to 1, we know that the normal must be corrected at the first sampling point (essentially approximating some finitely thick surface).

In the implicit case, no such explicit 'discontinuous' flag exists; it must be inferred from the information available. An observation of the rendering process described in previous sections is that if a ray has just finished 'skipping' an object block and begins to traverse static data, then this data must be implicitly split and is therefore a candidate for normal correction. This identification is handled easily on the CPU, and informing the fragment shader is a case of sending a uniform boolean value into the shader that deals with the raycasting of static data.

Figure 7.14 gives a comparison between the normal correction techniques implemented in our system. Note that the only light in the scene is arriving from the bottom right of the image; pointing towards the viewer on the image-space z-axis. Figure 7.14 empirically shows the correctness of the discontinuous world scheme in this case, as the surface of the split is almost black due to receiving no light.

Now that methods for identifying discontinuous regions have been discussed, we next discuss the implementation of both normal blending and a more advanced normal correction scheme based on computing the normal in discontinuous world space.

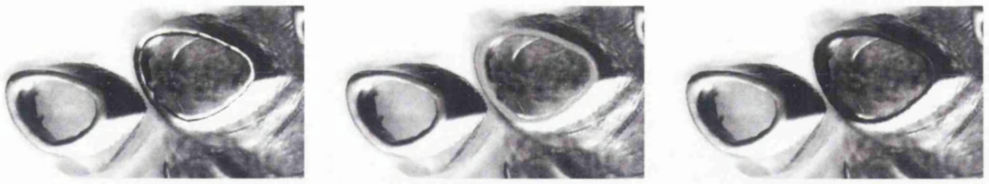

(a) No correction (b) Blended (c) Discontinuous World

Figure 7.14: Comparison between normal correction schemes

Implementing Normal Blending

The technique discussed above for blending the surface and volume normals can be implemented in our rendering algorithm as follows.

The volume normal N_{vol} can be computed as the forward-mapped world normal at the next ray sample point: $N_{vol} = \Phi_n(CentralDiff(p_0))$, and N_{dis} is the normal of the discontinuous face, i.e. the normal of the current fragment. With these two normals computed, equation 7.1 is applied with a suitable blending factor ω to produce the corrected normal. This normal is now used in the lighting equations to light the discontinuous region more accurately. Figure 7.14(b) shows the normal blending technique implemented in the system.

Implementing Discontinuous World Space Normal Computation

The technique of blending the normal from volume to surface produces a reasonable approximation of the discontinuous normal, but it remains only an approximation. We have the advantage in our wire cutting system that the block face representing the discontinuity is essentially a plane, and can thus be expressed in the form $ax + by + cz = 0$. Using this information, the normal at the discontinuity can be computed in world space by backward-mapping the central differencing sample points using Φ^{-1}, and correctly identifying those sample points which fall in empty space.

Figure 7.13(c) illustrates (in a 2D analogy) a ray beginning its journey from a discontinuous face, which has a normal of N_{dis}. Around the first ray sample point p_0 on the surface are the sample points required for the central differencing equation. To compute a correct normal N for the discontinuity, we use Equation 7.2:

$$\begin{aligned} N_x &= DisPoint(x+1,y,z) - DisPoint(x-1,y,z) \\ N_y &= DisPoint(x,y+1,z) - DisPoint(x,y-1,z) \\ N_z &= DisPoint(x,y,z+1) - DisPoint(x,y,z-1) \end{aligned} \tag{7.2}$$

where $DisPoint$ is a function that takes a point in the discontinuous world space $p \in \mathbb{E}^3$ and returns a correct sample point in object space. It works by returning a point inside the volume if and only if the point is deemed to be inside the wire block; i.e. if the point is 'past' the plane based on its outward-facing normal. If the point is deemed to be outside of the wire block, then a value of zero is returned to represent the empty space caused by the discontinuity:

$$DisPoint(P) = \begin{cases} \mathrm{V}(\Phi^{-1}(P)) & \text{if } N_{dis} \cdot P > -N_{dis} \cdot p_0 \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

where N_{dis} is the discontinuous face normal and p_0 is the initial ray sample point on the face.

This method of computing the normals in discontinuous world space inherently produces more accurate normals since world space is being treated correctly as discontinuous; the alternative method of blending the normal from the interior to exterior plane is just an approximation.

7.7 Improvements and Optimisations

The rendering algorithm given so far gives high-quality results for rendering the deformations, yet there is still room for improvement when one considers the possibility of supporting intersections (Section 7.7.1), and providing an important optimisation using the stencil buffer (Section 7.7.2).

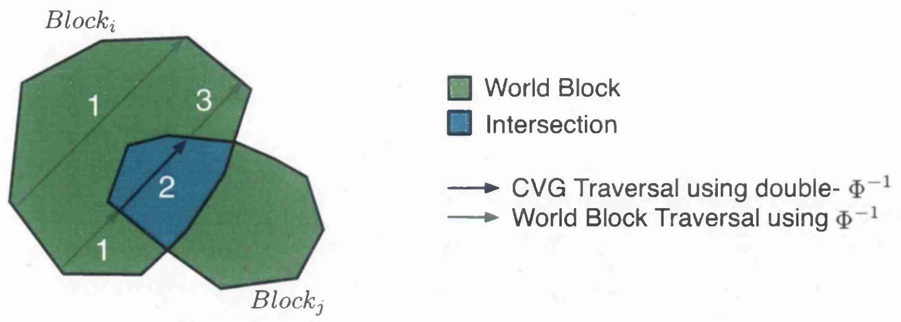
7.7.1 Future Support for Intersections

The rendering algorithm presented in Section 7.5 provided a simple priority system for world blocks – if two world blocks intersect, then the ray traverses completely through the first (in the depth order), and then continues through the second from the point of termination. This method does not allow for the intersection of two segmented subvolumes; inside the intersection, only one of the blocks is chosen for including in compositing operations. In a future implementation of the tool, it would be beneficial to provide intersection support so that segmented subvolumes can coexist in the same space. This section discusses a possible implementation of such functionality.

An example intersection is illustrated in Figure 7.15, which shows two world blocks intersecting (the blue area) and two rays travelling through. $Block_i$ is the currently-processed block in the main loop, and $Block_j$ is a block deemed to be intersecting. The leftmost ray does not touch the intersection area, but the rightmost ray travels partway through the world block (stage 1), then through the intersection (stage 2), and finally back through the world block (stage 3).

An algorithm for the implementation of intersections is given in Algorithm 7.

- *Stage one* raycasts (with Φ^{-1}) through $Block_i$ up until either the back faces or the front face of $Block_j$
- *Stage two* raycasts (with CVG and double-Φ^{-1}) the intersection area
- *Stage three* raycasts (with Φ^{-1}) through the remainder of $Block_i$ (if it exists)



$Block_i$
1
3
2
1
$Block_j$
🟩 World Block
🟦 Intersection
CVG Traversal using double- Φ^{-1}
World Block Traversal using Φ^{-1}

Figure 7.15: Dealing with the intersection of two world blocks

Raycasting the Intersection

An observation of intersections is that the rays in such areas are defined as having a starting depth greater than the termination depth. To render such an intersection of two world blocks therefore, the fragment shader can be instructed to fire rays from the termination point to the starting points. The fragment shader can perform two evaluations of Φ^{-1} at each ray sample point (halving the $STEP\_SIZE$ to prevent the maximum instruction limit being hit) and use the CVG [CT00] intersection operator to combine the values.

Either of the two blocks can be rasterised to initiate the rays, but the CVG raycasting shader must have access to the interpolated t-value of both blocks for each ray. In addition, the t-values will both need to be refined inside the loop and saved for retrieval in the next pass. Since the compositing buffer is fully utilised, the ray termination buffer's α component can be used for this purpose. This requires the Multiple Render Target (MRT) functionality of modern graphics hardware, which allows for up to four buffers to be written to in a single pass by writing values to the available colour output semantics.

Before the intersection is raycasted, $Block_j$ is rasterised, and its interpolated t-values are written to Ray$_{term}$'s α components. The CVG raycasting shader is then instantiated by rasterising $Block_i$. Because each block will potentially have a different wire ID, the wire ID of $Block_j$ is passed into the shader as a uniform variable. At each ray step, Φ^{-1} is performed for both blocks, the final value derived for each sample point using CVG's intersection operator, and the result composited. Because two blocks are being raycast, their t-values must both be refined before the ray loop iterates. Before exiting, the shader writes the refined t-value for $Block_i$ to the compositing buffer's α value (packed as a 16-bit float), and $Block_j$'s refined t-value back into Ray$_{term}$'s α value.

7.7.2 Using the Stencil Buffer for Increased Speed

A simple optimisation that can be made in the system is that of optimising the number of rays sent through the scene for every frame, avoiding raycasting areas of the image that are unchanged from frame to frame. During a deformation session, two events can prompt the renderer to draw a new frame:

- *Camera Adjustment* – in this case, new rays must be fired through the world based on the new viewing parameters to discover the true data.
- *Deformation of wires* – since the user only deforms one object at a time, only the portions of the image affected by the deformed object are required to be redrawn. The results for all other pixels (and consequently, rays) can be kept in the framebuffer.

Note that the deformation optimisation identified here assumes that the deformation of an object within the scene has no effect on other portions of the volume – i.e. there are no secondary rays being fired from the object for shadow computations or global illumination. Such a restriction however could be alleviated slightly by redrawing the complete scene (with information from the secondary rays) once the user has released control of the deformed object.

The aim of the optimisation is to correctly identify, for each frame, the rays that are required to be cast into areas that have changed. In the coming text, a pixel refers to a pixel in the framebuffer that is associated with a set of rays; many blocks may generate many rays, but the change of just one sample along the ray will ultimately cause a different composited colour.

For the scenario where the user is manipulating the wires (not changing the viewing parameters), a change in pixel value can occur in two areas:

- A pixel affected directly by a wire block being actively moved behind it; and,
- A static pixel directly affected by a wire block in the previous frame.

In both cases, all rays generated by wire blocks for those 'dirty' pixels should be re-fired. The detection of dirty pixels and subsequent optimisation is achieved using the stencil buffer functionality of the API and hardware. Because Framebuffer Objects are used, a stencil buffer is required to be attached to the bound framebuffer object, via the *packed_depth_stencil* extension which internally divides the depth buffer into a 24-bit depth value and and 8-bit stencil value.

The main rendering algorithm utilises the fast stencil rejection ability of the graphics hardware to ensure that only image areas made invalid by user operations are computed. If the viewing parameters are changed, the stencil buffer is cleared to 1 and the entire scene is redrawn, as a change in viewing parameters automatically implies that each pixel will be made invalid.

A new stage is introduced into the rendering pipeline to deal with the stencil buffer, prior to the initial probe stage. When the user modifies a world wire, its corresponding world blocks have an *update* flag set to denote that the block is being actively deformed in world space. Each frame has the previous frame's stencil buffer as input; at the beginning of a frame, all updated blocks from the previous frame will have set the stencil pixels to 1, though in the case of a change in viewing parameters, the stencil buffer is cleared to 1 at the end of the frame.

All blocks that are actively being deformed by the user (with the *update* flag set) are first rasterised, resetting the compositing buffer's $< R, G, B, A >$ values to 0 (and consequently, the new ray's alpha value to 0) and the stencil buffer to 1. The stencil buffer now contains

1 where the updated blocks for the current frame were rasterised. In addition, the stencil buffer will also contain 1 where an updated block existed in the previous frame.

The initial probe stage now fires rays into the scene where the stencil buffer is equal to 1. The main block loop (detailed in Algorithm 5) rasterises only blocks that are connected to blocks with the *update* flag set in the depth graph, and only rays belonging to stencilled pixels with value 1 are fired.

Before the rendering algorithm completes, the stencil buffer is cleared to 0, and all active blocks are rasterised with no colour writes, setting the new stencil buffer to 1 where these blocks are to be updated, ready for the next frame. The block's *update* flag is finally removed.

7.8 Summary

This chapter has introduced a complete tool for volume deformation. The tool not only offers the interactive specification of deformations, but also offers a real-time raycasting algorithm that evaluates the deformation in world space on-the-fly, without generating intermediate data. The deformation methodology implemented is the Volume Wires methodology, which is discussed in previous chapters of this thesis. The user specifies the deformation by adding object wires to the scene and using built-in segmentation functionality to define which portion of the volume is to be controlled by the wire. The segmentation functionality introduced is based on energy-minimising Snakes and uses a slice-based approach to build a 3D representation of the segmentation result from a sequence of 2D snakes defined along the trajectory of the wire. The tool therefore not only facilities deformation, but also the addition of semantic information necessary for character-based deformation.

The rendering algorithm employed by the system provides a complete, real-time fully raycasted view of the deformed object. The scene in world space is defined by the union of all segmented subvolumes and the static, unsegmented volume data. The algorithm exploits the segmentation information defined by the user to generate the starting and termination points for those rays that must be fired through the deformed subvolumes in world space. The set of all subvolumes (in object space) and forward-mapped subvolumes (in world space) defined in the scene are depth-sorted and rasterised in depth-order in order to progress rays through the scene; the algorithm can therefore be viewed as a hybrid object / image-space rendering algorithm.

The rendering algorithm introduced for the deformation has been accompanied by a discussion on the correct computation of normals in the case that world space has become discontinuous, and details on the implementation of normal correction techniques. Finally, two improvements to the tool have been discussed; including a method for rendering inside the intersection of deformed subvolumes, and a method for improving the interactivity while the user is deforming a wire by selectively raycasting only the parts of the image that will change as a result of the interaction.

Dataset	Action	Average FPS
Visman Torso	Navigation	8.44
	Manipulation	8.59
CT Carp	Navigation	2.63
	Manipulation	3.05
Tooth	Navigation	6.06
	Manipulation	6.29
Lobster	Navigation	8.72
	Manipulation	7.66

Table 7.1: A selection of timings achieved with four datasets. For each dataset, we give the average FPS achieved while firstly simply navigating the deformed dataset, and secondly while manipulating the world wires.

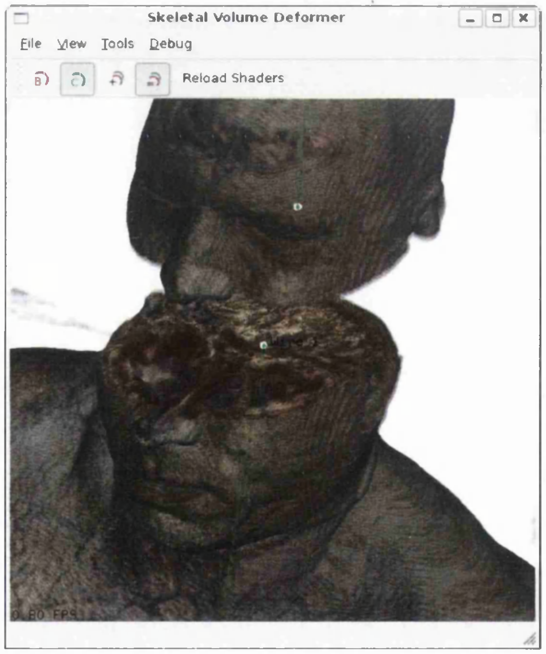


Figure 7.16: A split of the Visible Human's head. A full RGB colour texture was registered with the CT data and used as the transfer function, which reduced the rendering speed (≈ 3 FPS).

Algorithm 5 Wire Block Rasterisation

```
for $i : 1 \rightarrow SortedBlockList.size()$ do
  if $Block_i$ is a World Block then
    {Generate ray entry points for the world block}
    DrawBuffer(Ray$_{entry}$)
    Depth : 1, DepthFunc : $\leq$
    Rasterise front faces of $Block_i$
    DepthFunc : $>$
    for all World Blocks $Block_j$ obscuring $Block_i$ do
      Rasterise back faces of $Block_j$
    end for
    {Generate ray termination points for the world block}
    DrawBuffer(Ray$_{term}$)
    Depth : 1, DepthFunc : $\geq$
    Rasterise back faces of $Block_i$
    Depth : 0, DepthFunc : $\leq$
    DrawBuffer(Compositing)
    Raycast (with $\Phi^{-1}$) Ray$_{entry}$ $\rightarrow$ Ray$_{term}$
  end if
  {Generate ray starting points for possible static data}
  DrawBuffer(Ray$_{entry}$)
  Depth : 0, DepthFunc : $\geq$
  Rasterise back faces of $Block_i$
  for all Object Blocks $Block_j$ obscuring $Block_i$ do
    Rasterise back faces of $Block_j$
  end for
  {Generate ray termination points for the object block}
  DrawBuffer(Ray$_{term}$)
  Depth : 1, DepthFunc : $\leq$, DepthMask : $false$
  Rasterise back faces of Volume Dataset
  DepthMask : $true$
  for all Blocks $Block_j$ behind $Block_i$ do
    Rasterise front faces of $Block_j$
  end for
  DrawBuffer(Compositing)
  Raycast Ray$_{entry}$ $\rightarrow$ Ray$_{term}$
end for
```

Algorithm 6 GPU Backward-Mapping

```
ray = FromRayTextures();
finalColour = FromCompositingTexture();

if CPURayStep == 0 then
    WorldWirePoint = RefineWirePoint(FromFragmentColour());
else
    WorldWirePoint = RefineWirePoint(FromAlphaComponent());
end if

for all ray sample points $p_i$ do
    samplePt = $\Phi^{-1}$();
    sampleColour = TransferFunction(SampleVolume(samplePt));
    if sampleColour.$\alpha > 0$ then
        normal = NormalToWorld(CentralDifferences(samplePt));
        finalColour += composite();
    end if
    if finalColour.$\alpha > 0$ then
        break;
    end if
    WorldWirePoint = RefineWirePoint();
end for
```

Algorithm 7 Intersection Rendering

```
{Stage One}
Generate ray entry points as per Algorithm 5
Generate ray termination points from the back of $Block_i$ and front of intersecting block
Rasterise $Block_i$ and raycast (with $\Phi^{-1}$) Ray$_{entry} \rightarrow$ Ray$_{term}$
Generate ray entry points
{Stage Two}
Rasterise $Block_j$ and write $t$-values to Ray$_{term}$'s $\alpha$ component
Rasterise $Block_i$ and raycast (with CVG-$\Phi^{-1}$) Ray$_{term} \rightarrow$ Ray$_{entry}$
{Stage Three}
Generate ray entry points as back face of intersecting block
Generate ray termination points as back face of $Block_i$
Rasterise $Block_i$ and raycast (with $\Phi^{-1}$) Ray$_{entry} \rightarrow$ Ray$_{term}$
```

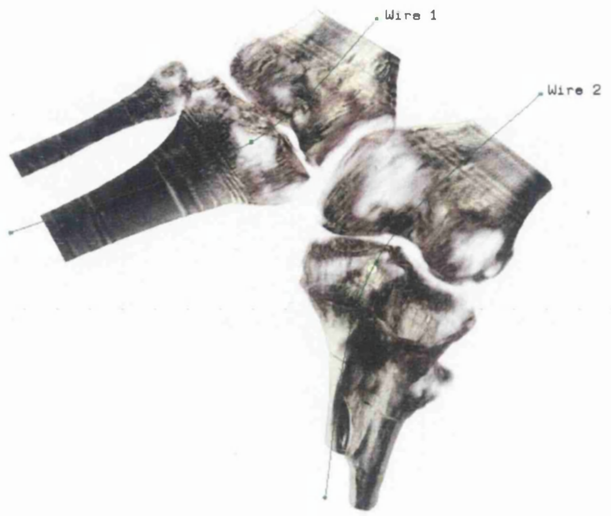


Figure 7.17: Manipulation of the CT Knee dataset ($\approx$ 4 FPS).

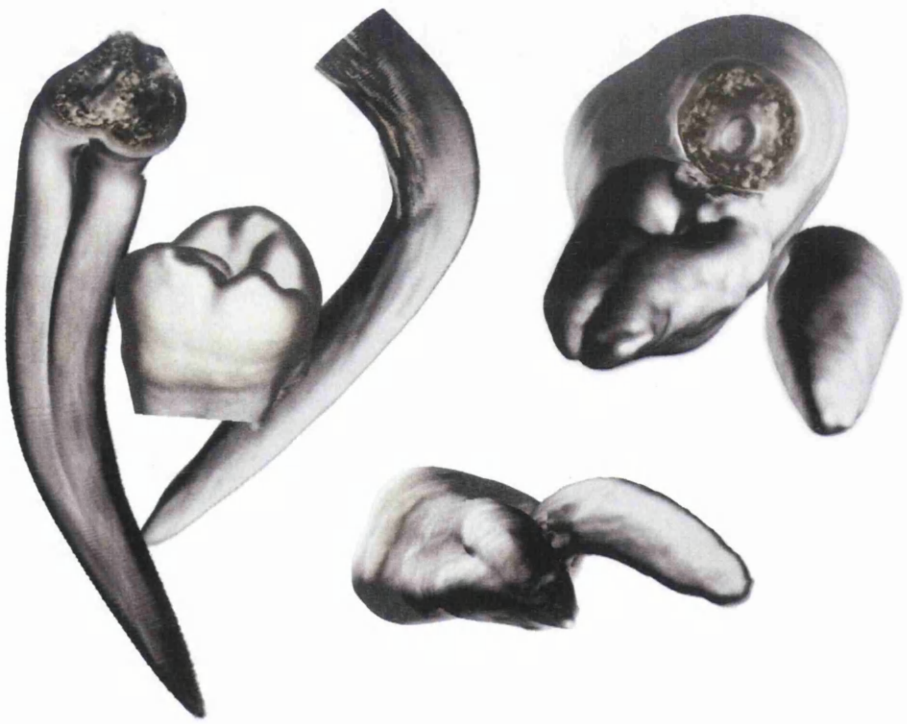


Figure 7.18: A series of artistic deformations with the tooth dataset. The roots have been bent around into new shapes, and also pulled away from the body of the tooth to separate them, without invoking the splitting functionality ($\approx$6 FPS average).

Chapter 8

Conclusions

Contents

The objectives of this thesis, as stated in Chapter 1, are:

1. To develop a methodology and framework for the deformation of volumetric datasets that will enable users to produce global deformations in an intuitive manner. The framework must support the use of traditional raycasting algorithms.
2. To investigate the feasibility of forward and backward mapping approaches for volume deformation in real-time.
3. To study visual enhancements to volume rendering through the use of global illumination techniques.
4. To provide a complete tool for intuitive and interactive volume deformation.

In this final chapter, the contributions of this thesis are summarised in the context of the thesis objectives.

Parts of this work have been published internationally:

- Parts of the work contained in Chapter 5 was presented at the Winter School of Computer Graphics (WSCG) 2006, and additionally accepted for publication in the Journal of the Winter School of Computer Graphics [WJ06].
- Parts of the work contained in Chapter 6 was presented at the Fourth International Conference on Medical Information Visualisation (MediViz) 2007 [WJ07].

8.1 A Volume Isosurface Renderer with Global Illumination

Chapter 4 introduced the area of GPU-based volume rendering to the reader, and later gave an implementation of a load-balancing renderer capable of combining both volume objects and surface-based objects in the same scene. To achieve this, an existing renderer called Igneus [Spe] was utilised that provides an extensible object-oriented interface for adding additional functionality.

Rays computed by Igneus are bundled together into contiguous memory blocks, and sent to the GPU using Framebuffer Objects with a 32-bit floating-point internal format. The GPU tests for intersections with the volume dataset (including any deformations enabled on the data), and returns a list of intersection depths, if any. While the GPU is computing the intersections, Igneus is able to perform any of its own computations (including intersections with surface-based data in the scene, and lighting computations) that do not depend on the result of the ray depths computed by the GPU. This provides an effective load-balancing scheme in that both the CPU and GPU are being worked concurrently as much as possible within the limitations of the ray tracer.

We have given an analysis of method when volume isosurface intersections are computed only on the CPU, and shown that the GPU-based method provides a significant improvement in rendering times. In addition, many example images from the system have been given, showing the extremely high-quality renderings that are possible when combining volume isosurfaces with an advanced ray tracing application that incorporates global illumination.

8.2 An Intuitive Volume Deformation Methodology and Framework

Chapter 5 introduced a volume deformation methodology and framework called *Volume Wires*; an initial study of which has been published in the Journal of the Winter School of Computer Graphics 2006 [WJ06]. A study of existing volume deformation methods and algorithms was given in Chapter 3, where it was shown that although much research has been conducted in the area, a comparatively small amount of research has gone into making intuitive and simple methodologies for character-based and globally-based deformations.

The Volume Wires methodology introduces the concept of using curve-skeletons to deform volume objects. The user first defines the curve skeletons in object space (the *object wires*), and then deforms these wires in world space (*world wires*) in order to deform the volume object. Curve-skeletons are not new in themselves; and nor is their application in volume deformation (see the works of Silver [GS99, GS00b, SSC03]). However, our method given in Chapter 5 defines *non-reconstructive* framework for such a methodology; as such, the volume dataset integrity is retained and only referred to via backward-mapping operations. The mapping operations used have been analytically detailed using example images from MATLAB.

A variety of example high-quality result images have been shown which highlight that the rendering stage for the framework can be any existing raycasting volume renderer capable of evaluating a backward-mapping function Φ^{-1} for each sample point on the ray. The images demonstrate not only the advantage of utilising a backward-mapping operation with full raycasting, but also the intuitiveness of the methodology.

8.3 GPU-based Forward-Projection Volume Deformation

During development of the Volume Wires methodology and framework, it become clear that an interactive implementation would demonstrate its effectiveness and intuitive nature. Chapter 6 introduced a GPU-based forward-projection algorithm for the Volume Wires methodology. Before implementation details were given, a detailed discussion was given on the issues surrounding forward-projection on the GPU; particularly when the data has been deformed prior to rendering. The issue of cracks appearing in the final image due to samples being pulled apart was also discussed, with possible object and image-space solutions.

The rendering algorithm introduced in the chapter utilised the GPU for almost all of the render-time computation, with the full Volume Wires forward-mapping operation being performed in the vertex shader on each incoming voxel encoded as a vertex. The advantage of using forward-projection for a volume deformation tool for the Volume Wires methodology immediately became apparent in that the mapping field was only required to be computed previously to the user deforming the world wires in the scene. Once the object wires were defined, the mapping field could be computed for the object wires, and an association set up between voxels in object space and their nearest wire offsets.

In addition, the integration of segmentation information (in the form of a mask volume) was detailed, with the only modification required to the system being in the mapping field generation stage.

8.4 A Complete, Raycasted Volume Deformation Tool

The final chapter of the thesis, Chapter 7, introduced a complete volume deformation tool based on the Volume Wires methodology and framework. The goal of the tool was to provide a complete, real-time, and interactive deformation tool that computed the Volume Wires backward-mapping function Φ^{-1} in *real-time*. The ability for the backward-mapping function to be computed in real-time provides a real-time raycasting approach to volume deformation.

The tool introduced in Chapter 7 provides the user with a real-time raycasted view of deformations achieved in the Volume Wires methodology, complete with the full deformed internal texture of the volume dataset. The software employs a built-in segmentation tool based on energy-minimising Snakes [KWT88]. Once the segmented subvolumes within the volume dataset are defined for each wire, the rendering algorithm is able to forward-map these subvolumes (defined as polygonal meshes) into world space and generate the ray entry/exit points. After depth-sorting, the blocks are internally raycasted.

We have shown from the resulting images that the rendering algorithm provides very a high-quality output, complete with full interior lighting computations (with normals forward-mapped into world space) and 32-bit compositing. In addition, an analysis has been given to the tool's splitting functionality, including a discussion on the computation of normals in areas of discontinuity.

8.5 Conclusions & Future Work

The main contribution of this thesis was to introduce the concept of Volume Wires to enable forward and backward mapping of volume deformations. The use of such a methodology has enabled continuous sampling of deformed space by backward mapping into object space. The merits of such a technique can be further studied including applications to other areas of discretely sampled object representations (for example: image morphing and point based rendering).

Three rendering algorithms for visualising deformations in the Volume Wires framework have been introduced. Chapter 5 introduced the Volume Wires framework, and gave an example GPU-based raycasting rendering algorithm that evaluated an indirection volume (the *mapping field*) in order to gain the necessary values for input into the backward-mapping function Φ^{-1}. Though the rendering algorithm was interactive, the process of generating the mapping field each time the world wires were modified did not facilitate interactive manipulation. Chapter 6 took a different approach, presenting a GPU forward-projection algorithm that required the mapping field to only be generated for the object wires; thus facilitating interactive manipulation during world wire manipulation. Though the interaction/rendering was interactive, a full blending of the samples could not be achieved due to a guaranteed image-space traversal being unfeasible to compute. Chapter 7 finally presented a method that combined the best of both these algorithms – forward-projecting the boundaries of the deformed model and internally raycasting these boundaries. This algorithm was shown to provide full internal raycasting of the Volume Wires mapping, with interactive frame rates in most cases.

Further studies for usability can be anticipated. In this work, the user is provided with a segmentation tool, with examples of segmenting the visible human arm and the ability to move it away from the body. This still relies on the semi-automatic placement and segmentation of snakes. Usability could be increased if the segmentation can be deduced from merely where the user is pointing. In this example, a segmentation algorithm could use the fact that the user is pointing to the arm to segment the most likely region (this is partially implemented by inferring the best depth by casting a ray through the data and choosing the mean depth of all non-empty data). Techniques such as the magnetostatic active contour method [XM07] may allow such a method.

As the line between CPU and GPU programming becomes less clear over time, new methods for volume rendering and volume deformation will become viable for GPU implementations. The work contained within this thesis uses the state-of-the-art hardware available *at the time*, but it is clear that the field will evolve significantly with the hardware capabilities. In particular, the work contained within Chapter 7 has much scope for further development as the harsh restrictions placed on GPU developers are lifted; paving the way for gradually more flexible algorithms running at higher speeds.

We are confident that this thesis has provided a firm foundation for future researchers wishing to develop volume rendering and deformation techniques on graphics hardware. It is an immensely rewarding and challenging field to be a part of.

Appendix A

Volume Wires XML Schema

Listing A.1: Volume Wires State XML Schema

```
<xs:element name="Point">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="X" type="xs:decimal"/>
         <xs:element name="Y" type="xs:decimal"/>
         <xs:element name="Z" type="xs:decimal"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="SnakeVertex">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="X" type="xs:decimal"/>
         <xs:element name="Y" type="xs:decimal"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="VolumeWiresState">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="TargetVolume">
            <xs:complexType>
               <xs:sequence>
                  <xs:attribute name="Filename" type="xs:string"/>
                  <xs:element name="DataSizeX" type="positiveInteger"/>
                  <xs:element name="DataSizeY" type="positiveInteger"/>
                  <xs:element name="DataSizeZ" type="positiveInteger"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         <xs:element name="Wires">
            <xs:element name="Wire" maxOccurs="unbounded">
```

```
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="Base">
                        <xs:element ref="Point" maxOccurs="unbounded"/>
                     </xs:element>
                     <xs:element name="Control">
                        <xs:element ref="Point" maxOccurs="unbounded"/>
                     </xs:element>
                     <xs:element name="SliceList">
                        <xs:attribute name="MaxCount" type="xs:Integer"/>
                        <xs:attribute name="PlaneSize" type="xs:Integer"/>
                        <xs:element name="Slice" maxOccurs="unbounded">
                           <xs:element ref="SnakeVertex" maxOccurs="unbounded"/>
                        </xs:element>
                     </xs:element>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:element>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Bibliography

[3ds] Autodesk 3d studio max. `http://www.autodesk.com/3dsmax`.

[AF97] Mohamed N. Ahmed and Aly A. Farag. Two-stage neural network for volume segmentation of medical images. *Pattern Recogn. Lett.*, 18(11-13):1143–1151, 1997.

[AMBJ02] K. Adbel-Malek, D. Blackmore, and K. Joy. Swept volumes: Foundations, perspectives, and applications. In *International Journal of Shape Modeling*, 2002.

[AMYO98] Karim Abdel-Malek, Harn-Jou Yeh, and Saeb Othman. Swept volumes: void and boundary identification. *Computer-Aided Design*, 30(13):1009–1018, 1998.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[ARC05] Alfie Abdul-Rahman and Min Chen. Spectral volume rendering based on the kubelka-munk theory. *Computer Graphics Forum*, 24:413–422, 2005.

[ASK94] R. S. Avila, L. M. Sobierajski, and A. E. Kaufman. Visualizing nerve cells. 14(5):11–13, September 1994.

[Bar84] Alan H. Barr. Global and local deformations of solid primitives. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 21–30, New York, NY, USA, 1984. ACM Press.

[Bar86] Alan H. Barr. Ray tracing deformed surfaces. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1986. ACM Press.

[Bar92] Ronen Barzel. *Physically-Based Modeling for Computer Graphics*. Academic Press, Inc., 1992.

[BB03] L. Ballerini and L. Bocchi. Bone segmentation using multiple communicating snakes. In M. Sonka and J. M. Fitzpatrick, editors, *Medical Imaging 2003: Image Processing. Edited by Sonka, Milan; Fitzpatrick, J. Michael. Proceedings of the SPIE, Volume 5032, pp. 1621-1628 (2003).*, pages 1621–1628, May 2003.

[BFGS03] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[BG06] Stefan Bruckner and Meister Eduard Gröller. Exploded views for volume data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1077–1084, 2006.

[BGB$^+$06] Kevin M. Beason, Josh Grant, David C. Banks, Brad Futch, and M. Yousuff Hussaini. Pre-computed illumination for isosurfaces. In *Visualization and Data Analysis 2006 (SPIE Vol. 6060)*, pages 6060B:1–11, 2006.

[BHZK05] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings of the Eurographics Symposium on Point-Based Graphics*, pages 17–24, June 2005.

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern GPUs. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, pages 335–343. IEEE Computer Society, 2003.

[Blo85] Jules Bloomenthal. Modeling the mighty maple. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 305–311, New York, NY, USA, 1985. ACM Press.

[Blo90] Jules Bloomenthal. Calculcation of reference frames along a space curve. In Andrew S. Glassner, editor, *Graphics Gems*, chapter 10, pages 567–571. Academic Press Professional, 1990.

[Blu67] H. Blum. A Transformation for Extracting New Descriptors of Shape. In *Proceedings of the Symposium on Models for the Perception of Speech and Visual Form*, pages 362–380, 1967.

[BLWJ97] D. Blackmore, M. C. Leu, L. P. Wang, and H. Jiang. Swept volume: a retrospective and prospective view. *Neural, Parallel Sci. Comput.*, 5(1-2):81–102, 1997.

[BM99] David E. Breen and Sean Mauch. Generating shaded offset surfaces with distance, closest-point and color volumes. In *Proceedings of the International Workshop on Volume Graphics*, pages 307–320, March 1999.

[BNC96] Morten Bro-Nielsen and Stephane Cotin. Real-time volumetric deformable models for surgery simulation using finite elements and condensation. *Computer Graphics Forum*, 15(3):57–66, 1996.

[BPS96] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Fast isocontouring for improved interactivity. In *VVS '96: Proceedings of the 1996 sym-*

posium on Volume visualization, pages 39–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[BSKS05] A. Barsi, L. Szirmay-Kalos, and G. Szijártó. Stochastic glossy global illumination on the GPU. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 187–193, New York, NY, USA, 2005. ACM Press.

[Car84] Loren Carpenter. The A -buffer, an antialiased hidden surface method. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 103–108, New York, NY, USA, 1984. ACM Press.

[Cat75] Edwin E. Catmull. Computer display of curved surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975.

[CC05] Steven P. Callahan and Joao L. D. Comba. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005. Student Member-Milan Ikits and Member-Claudio T. Silva.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM Press.

[CCS06] Min Chen, Carlos Correa, and Deborah Silver. Feature aligned volume manipulation for illustration and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1069–1076, 2006.

[CDA00] Stéphane Cotin, Hervé Delingette, and Nicholas Ayache. A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation. In *The Visual Computer*, volume 16(8), pages 437–452. Springer, 2000.

[CE98] Jegathese CR and Prakash EC. Goal-directed deformation of the visible human. In *Proceedings of the Visible Human Project Conference*, Maryland, USA, October 1998.

[CH02] Liviu Coconu and Hans-Christian Hege. Hardware-accelerated point-based rendering of complex scenes. In Simon Gibson and Paul Debevec, editors, *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 43–52, Pisa, Italy, June 2002.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CHH03] Nathan A. Carr, Jesse D. Hall, and John C. Hart. GPU algorithms for radiosity and subsurface scattering. In *HWWS '03: Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 51–59, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[CHP89] J. E. Chadwick, D. R. Haumann, and R. E. Parent. Layered construction for deformable animated characters. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 243–252, New York, NY, USA, 1989. ACM Press.

[CHRU85] Lih-Shyang Chen, Gabor T. Herman, R. Anthony Reynolds, and Jayaram K. Udupa. Surface shading in the Cuberille environment. 5(12):33–43, December 1985.

[CIJ+07] M. Chen, S. Islam, M. W. Jones, P.-Y. Shen, D. Silver, S. J. Walton, and P. J. Willis. Manipulating, deforming and animating sampled object representations. *Computer Graphics Forum*, 26(4):824–852, December 2007. Initially presented in Eurographics State of the Art Report in 2005.

[CLW04] J. Cates, A. Lefohn, and R. Whitaker. Gist: an interactive, gpu-based level set segmentation tool for 3d medical images. Technical Report UUCS-04-007, University of Utah School of Computing, 2004.

[CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, Chapel Hill, NC, USA, 1994.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[Coq90] Sabine Coquillart. Extended free-form deformation: a sculpturing tool for 3d geometric modeling. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 187–196, New York, NY, USA, 1990. ACM Press.

[Cor03] NVIDIA Corporation. Using vertex buffer objects, 2003. Whitepaper.

[CRZP04] Wei Chen, Liu Ren, Matthias Zwicker, and Hanspeter Pfister. Hardware-accelerated adaptive EWA volume splatting. In *Proceedings of IEEE Visualization 2004*, October 2004.

[CS94] Daniel Cohen and Zvi Sheffer. Proximity clouds–an acceleration technique for 3d grid traversal. *Vis. Comput.*, 11(1):27–38, 1994.

[CSC03] Carlos D. Correa, Deborah Silver, and Min Chen. Discontinuous Displacement Mapping for Volume Graphics. In *Proc. Volume Graphics 2003*, pages 9–16, 2003.

[CSM05] Nicu D. Cornea, Deborah Silver, and Patrick Min. Curve-skeleton applications. In *IEEE Visualization*, pages 95–102, 2005.

[CSW+03] M. Chen, D. Silver, A. S. Winter, V. Singh, and N. Cornea. Spatial transfer functions – a unified approach to specifying deformation in volume modeling and animation. In *Proc. Volume Graphics 2003*, pages 35–44, Tokyo, Japan, 2003.

[CT00] Min Chen and John V. Tucker. Constructive volume geometry. In David Duke, Sabine Coquillart, and Toby Howard, editors, *Computer Graphics Forum*, volume 19(4), pages 281–293. Eurographics Association, 2000.

[CUD] NVIDIA developer : CUDA. `http://developer.nvidia.com/object/cuda.html`.

[CWRT02] Min Chen, Andrew S. Winter, David Rodgman, and Steven M. F. Treavett. Enriching volume modelling with scalar fields. In F. Post, G.-P. Bonneau, and G. Nielson, editors, *Data Visualization: The State of The Art*, pages 345–362. Kluwer Academic Press, 2002.

[Dam71] R. Damadian. Tumor detection by nuclear magnetic resonance. *Science*, 171:1151–1153, 1971.

[Dan80] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[DDCB01] Gilles Debunne, Mathieu Desbrun, Marie-Paule Cani, and Alan H. Barr. Dynamic real-time deformations using space and time adaptive sampling. In *Computer Graphics Proceedings*, Annual Conference Series. ACM Press / ACM SIGGRAPH, Aug 2001. Proceeding.

[Des03] Foraker Design. Usability first, usability glossary, 2003. `http://www.infovis-wiki.net/index.php/Focus-plus-Context`.

[DK00] Frank DachilleIX and Arie Kaufman. GI-cube: an architecture for volumetric global illumination and rendering. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 119–128, New York, NY, USA, 2000. ACM Press.

[DKC$^+$98] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufman. High-quality volume rendering using texture mapping hardware. In *HWWS '98: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 69–ff., New York, NY, USA, 1998. ACM Press.

[EPO91] A. B. Ekoule, F. C. Peyrin, and C. L. Odet. A triangulation algorithm from arbitrary shaped multiple planar contours. *ACM Trans. Graph.*, 10(2):182–199, 1991.

[EYSK94] D. S. Ebert, R. Yagel, J. Scott, and Y. Kurzion. Volume rendering methods for computational fluid dynamics visualization. In *Visualization'94*, pages 232–239, 1994.

[FKU77] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Commun. ACM*, 20(10):693–702, 1977.

[FRZ$^+$04] Daniel Freedman, Richard J. Radke, Tao Zhang, Yongwon Jeong, and George T. Y. Chen. Model-based multi-object segmentation via distribution matching. In *CVPRW '04: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04) Volume 1*, page 11, Washington, DC, USA, 2004. IEEE Computer Society.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice*. The Systems Programming Series. Addison-Wesley, second edition in c edition, 1996.

[Gag97] Nikhil Gagvani. Skeletons and volume thinning in visualization. Master's thesis, Graduate School, Rutgers, The State University of New Jersey, 1997.

[GEL] Ge logiq ultrasound scanning. `http://www.gehealthcare.com/usen/ultrasound/reimagined/usri_volume_landing.html`.

[GFG04] Philipp Gerasimov, Randima Fernando, and Simon Green. *Using Vertex Textures*. NVIDIA Corp., 2004. Whitepaper, ftp://download.nvidia.com/developer/Papers/.

[GH91] Tinsley A. Galyean and John F. Hughes. Sculpting: An interactive volumetric modeling technique. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 267–274, July 1991.

[GHF90] C. Giertsen, A. Halvorsen, and P. R. Flood. Graph-directed modelling from serial sections. *Vis. Comput.*, 6(5):284–290, 1990.

[Gib97] S. Gibson. 3D chainmail: a fast algorithm for deforming volumetric objects. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 149–154, April 1997.

[GK96] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 23–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[GKHS98] Nikhil Gagvani, D. Kenchammana-Hosekote, and D. Silver. Volume animation using the skeleton tree. In *IEEE Symposium on Volume Visualization*, pages 47–53, 1998.

[GLM04] Naga K. Govindaraju, Ming Lin, and Dinesh Manocha. Vis-sort: Fast visibility ordering of 3-d geometric primitives. Technical report, University of North Carolina at Chapel Hill, 2004.

[GM05] James Gain and Patrick Marais. Warp sculpting. *IEEE Transactions on Visualization and Computer Graphics*, 11(2):217–227, 2005. Member-James Gain and Member-Patrick Marais.

[GMA$^+$04] V. Grau, A.U.J. Mewes, M. Alcaniz, R. Kikinis, and S.K. Warfield. Improved watershed transform for medical image segmentation using prior information. 23(4):447–458, April 2004.

[GS99] Nikhil Gagvani and Deborah Silver. Parameter-controlled volume thinning. *CVGIP: Graph. Models Image Process.*, 61(3):149–164, 1999.

[GS00a] N. Gagvani and D. Silver. Animating the visible human dataset. In *NLM Visible Human Conference*, October 2000.

[GS00b] N. Gagvani and D. Silver. Realistic volume animation with alias. In Min Chen, editor, *Volume Graphics*, pages 253–263. Springer, 2000.

[GS01] Nikhil Gagvani and Deborah Silver. Animating volumetric models. *Graph. Models*, 63(6):443–458, 2001.

[GW05] Joachim Georgii and Rüdiger Westermann. Mass-spring systems on the gpu. *Simulation Modelling Practice and Theory*, 13:693–702, 2005.

[Har] Aaron Harwood. Parallel algorithms. `http://www.cs.mu.oz.au/498/notes/node40.html`.

[Har05] Mark Harris. Mapping computational concepts to GPUs. In Randima Fernando, editor, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 31, pages 493–508. Addison-Wesley, 2005.

[HB86] K.H. Höhne and R. Bernstien. Shading 3D images from CT using grey level gradients. *IEEE Transactions on Medical Imaging*, 5(1):45–47, March 1986.

[HBH03] Markus Hadwiger, Christoph Berger, and Helwig Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 40, Washington, DC, USA, 2003. IEEE Computer Society.

[HE99] Matthias Hopf and Thomas Ertl. Accelerating 3d convolution using graphics hardware (case study). In *VIS '99: Proceedings of the conference on Visualization '99*, pages 471–474, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[HE00] M. Hopf and T. Ertl. Accelerating Morphological Analysis with Graphics Hardware. In *Workshop on Vision, Modelling, and Visualization VMV '00*, pages 337–345. infix, 2000.

[HHCL01] Taosong He, Lichan Hong, Dongqing Chen, and Zhengrong Liang. Reliable path for virtual endoscopy: Ensuring complete examination of human organs. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):333–342, 2001.

[Hou73] Godfrey N. Hounsfield. Computerized transverse axial scanning (tomography): Part 1. description of system. *British Journal of Radiology*, 46:1016–1022, 1973.

[HP03] Horst Hahn and Heintz-Otto Peitgen. Iwt - interactive watershed transform: A hierarchical method for efficient interactive and automated segmentation of multidimensional grayscale images. *Proc. Medical Imaging,*, Feb 2003.

[HQK05] Wei Hong, Feng Qiu, and Arie Kaufman. GPU-based object-order ray-casting for large datasets. In Eduard Gröller and Issei Fujishiro, editors, *Eurographics/IEEE VGTC Workshop on Volume Graphics*, pages 177–185, Stony Brook, NY, 2005. Eurographics Association.

[IDSC04] Shoukat Islam, Swapnil Dipankar, Deborah Silver, and Min Chen. Temporal and spatial splitting of scalar fields in volume graphics. In *Proc. IEEE VolVis2004*, pages 87–94. IEEE, October 2004.

[IH] Milan Ikits and Charles Hansen. A focus and context interface for interactive volume rendering. Manuscript.

[imd] IMDB. http://www.imdb.com/.

[ISC07] Shoukat Islam, Deborah Silver, and Min Chen. Volume splitting and its applications. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):193–203, 2007.

[Isl07] Shoukat Islam. *Field-Based Volume Deformation.* PhD thesis, Swansea University, 2007.

[JB02] Henrik Wann Jensen and Juan Buhler. A rapid hierarchical rendering technique for translucent materials. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 576–581, New York, NY, USA, 2002. ACM Press.

[JBŠ06] Mark W. Jones, Andreas Bærentzen, and Miloš Šrámek. Discrete 3D distance fields: Techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):to appear, July/August 2006.

[JC94a] M. W. Jones and M. Chen. A new approach to the construction of surfaces from contour data. *Computer Graphics Forum*, 13(3):75–84, 1994.

[JC94b] Mark. W. Jones and Min Chen. Fast cutting operations on three dimensional volume datasets. In M. Göbel, H. Müller, and B. Urban, editors, *Visualization in Scientific Computing*, pages 1–8. Springer-Verlag Wien, May 1994.

[Jen04] Henrik Wann Jensen. A practical guide to global illumination using ray tracing and photon mapping. In *SIGRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, page 20, New York, NY, USA, 2004. ACM Press.

[JH99] P. Jaquays and B. Hook. Quake 3: Arena shader manual, revision 10. December 1999.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA, 2001. ACM Press.

[Jon95] Mark W. Jones. 3D distance from a point to a triangle. Technical Report CSR-5-95, Department of Computer Science, University of Wales, Swansea, February 1995.

[Jon96] Mark W. Jones. The production of volume data from triangular meshes using voxelisation. *Computer Graphics Forum*, 15(5):311–318, 1996.

[Jon97] M. W. Jones. An efficient shadow detection algorithm and the direct surface rendering volume visualisation model. In *Proc. 15th Ann. Conf. of Eurographics (UK Chapter)*, pages 237–244, 1997.

[Jon01] Mark W. Jones. Facial reconstruction using volumetric data. In *Vision, Modeling, and Visualization*, pages 135–142, 2001.

[Jon04] Mark W. Jones. Distance field compression. *Journal of WSCG*, 12(2):199–204, 2004.

[Kaj82] James T. Kajiya. Ray tracing parametric patches. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 245–254, New York, NY, USA, 1982. ACM Press.

[Kau87] Arie Kaufman. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 171–179, New York, NY, USA, 1987. ACM Press.

[KB04] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[KCY93] Arie Kaufman, Daniel Cohen, and Roni Yagel. Volume graphics. *Computer*, 26(7):51–64, July 1993.

[KEK03] Yan Kang, Klaus Engelke, and Willi A. Kalender. A new accurate and precise 3d segmentation method for skeletal structures in volumetric ct data. *IEEE Trans. Med. Imaging*, 22(5):586–598, 2003.

[KH84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 165–174, New York, NY, USA, 1984. ACM Press.

[KK99] Kevin Kreeger and Arie Kaufman. Interactive volume segmentation with the pavlov architecture. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, pages 61–68, New York, NY, USA, 1999. ACM Press.

[KPH$^+$03] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson. A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):150–162, 2003.

[KW03] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[KWPH06] A. Knoll, I. Wald, S.G. Parker, and C.D. Hansen. Interactive isosurface ray tracing of large octree volumes. SCI Institute Technical Report UUSCI-2006-026, University of Utah, 2006.

[KWT88] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1:321–331, 1988.

[KWW01] Davis King, Craig M. Wittenbrink, and Hans J. Wolters. An architecture for interactive tetrahedral volume rendering. In Klaus Mueller and Arie E. Kaufman, editors, *Volume Graphics*, volume Proceedings of the Joint IEEE TCVG and Eurographics Workshop on Volume Graphics in Stony Brook, New York, USA, 2001.

[KY95] Yair Kurzion and Roni Yagel. Space deformation using ray deflectors. In *Rendering Techniques '95 (Proceedings of the Sixth Eurographics Workshop on Rendering)*, pages 21–30, New York, 1995. Springer-Verlag.

[KY97] Yair Kurzion and Roni Yagel. Interactive space deformation with hardware-assisted rendering. *IEEE Comput. Graph. Appl.*, 17(5):66–77, 1997.

[Las87] John Lasseter. Principles of traditional animation applied to 3d computer animation. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 35–44, New York, NY, USA, 1987. ACM Press.

[LC87] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[LE98] Christoph L&#252;rig and Thomas Ertl. Hierarchical volume analysis and visualization based on morphological operators. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 335–341, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(3):29–37, May 1988.

[Lev90a] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[Lev90b] Marc Levoy. Volume rendering by adaptive refinement. *Vis. Comput.*, 6(1):2–7, 1990.

[Lib] Libsh. http://libsh.org/.

[Lif] Lifescan. http://www.lifescanuk.org/.

[LJT01] Zheng Lin, Jesse S. Jin, and Hugues Talbot. Unseeded region growing for 3d image segmentation. In Peter Eades and Jesse Jin, editors, *Selected papers from Pan-Sydney Area Workshop on Visual Information Processing*, volume 2 of *CRPIT*, pages 31–37, Sydney, Australia, 2001. ACS.

[LK04] Sarang Lakare and Arie Kaufman. Light weight space leaping using ray coherence. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 19–26, Washington, DC, USA, 2004. IEEE Computer Society.

[LKHW04] Aaron E. Lefohn, Joe Michael Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Trans. Vis. Comput. Graph.*, 10(4):422–433, 2004.

[LW85] M. Levoy and T. Whitted. The use of points as display primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, January 1985.

[Max95] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[May] Autodesk maya. `http://www.autodesk.com/maya`.

[MBL+91] James V. Miller, David E. Breen, William E. Lorensen, Robert M. O'Bara, and Michael J. Wozny. Geometrically deformed models: a method for extracting closed geometric models form volume data. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 217–226, New York, NY, USA, 1991. ACM Press.

[MDHK01] Michael Meißner, Mike Doggett, Johannes Hirche, and Urs Kanus. Efficient space leaping for ray casting architectures. In *Volume Graphics*, Workshop on Volume Graphics, pages 149–161, Stony Brook, NY, USA, June 2001.

[ME05] Benjamin Mora and David S. Ebert. Low-complexity maximum intensity projection. *ACM Trans. Graph.*, 24(4):1392–1416, 2005.

[Mey94] D. Meyers. Reconstruction of surfaces from planar contours, 1994.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[MHC90] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 27–33, New York, NY, USA, 1990. ACM Press.

[MJ05] C. M. Miller and M. W. Jones. Texturing and Hypertexturing of Volumetric Objects. In *Proceedings of the International Workshop on Volume Graphics '05*, pages 117–125, 2005.

[MJC02] Benjamin Mora, Jean Pierre Jessel, and Ren&#233; Caubet. A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 203–210, Washington, DC, USA, 2002. IEEE Computer Society.

[MKG00] Lukas Mroz, Andreas König, and Eduard Gröller. Maximum intensity projection at warp speed. *Computers and Graphics*, 24(3):343–352, 2000.

[MN88] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 221–228, New York, NY, USA, 1988. ACM Press.

[MTB03] Michael J. McGuffin, Liviu Tancau, and Ravin Balakrishnan. Using deformations for browsing volumetric data. In *Proceedings of IEEE Visualization (VIS) 2003*, pages 401–408, October 2003.

[Mul92] James C. Mullikin. The vector distance transform in two and three dimensions. *CVGIP: Graph. Models Image Process.*, 54(6):526–535, 1992.

[MV98] J. Maintz and M. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.

[MY96] Klaus Mueller and Roni Yagel. Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 65–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[Nad00] David R. Nadeau. Volume scene graphs. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 49–56, New York, NY, USA, 2000. ACM Press.

[NM05] Neophytos Neophytou and Klaus Mueller. Gpu accelerated image aligned splatting. In Arie E. Kaufman, Klaus Mueller, Eduard Gröller, Dieter W. Fellner, Torsten Möller, and Stephen N. Spencer, editors, *Volume Graphics*, pages 197–205, 2005.

[NNS72] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. In *ACM'72: Proceedings of the ACM annual conference*, pages 443–450, New York, NY, USA, 1972. ACM Press.

[NT98] L. P. Nedel and D. Thalmann. Real time muscle deformations using mass-spring systems. In *CGI '98: Proceedings of the Computer Graphics International 1998*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[NVI05] NVIDIA Corp. *NVIDIA GPU Programming Guide (version 2.4.0)*, 2005. http://developer.nvidia.com/object/gpu_programming_guide.html.

[obg] The history of ultrasound: A collection of recollections, articles, interviews and images. `http://www.obgyn.net/ultrasound/?page=/us/news_articles/ultrasound_history/asp-history-toc`.

[Ope01] OpenGL Architecture Review Board. *NV_point_sprite extension specification*, 2001. http://www.opengl.org/registry/specs/NV/point_sprite.txt.

[Owe05] John Owens. Streaming architectures and technology trends. In Randima Fernando, editor, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 29, pages 457–470. Addison-Wesley, 2005.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, 2002.

[PC98] Edmond C. Prakash and Jegathese CR. A new approach for goal-oriented deformation of voxel models. In *Proceedings of Pacific Graphics 1998*, pages 214–215, Singapore, October 1998.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253, July 1984. Proceedings of SIGGRAPH 84.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, New York, NY, USA, 1989. ACM Press.

[Phi] Philips brilliance ct. `http://www.medical.philips.com/main/products/ct/products/brilliance/index.html`.

[Piq90] Micheal E. Pique. Rotation tools. In Andrew S. Glassner, editor, *Graphics Gems*, chapter 11, pages 607–611. Academic Press Professional, 1990.

[Pix05] Pixar. *The RenderMan Interface, V3.2.1.* 2005. https://renderman.pixar.com/products/rispec/.

[PM90] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629–639, 1990.

[POAU00] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 425–432, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[PPL+99] S. Parker, M. Parker, Y. Livnat, P.P. Sloan, C.D. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[PT90] Bradley A. Payne and Arthur W. Toga. Surface mapping brain function on 3D models. *IEEE Computer Graphics and Applications*, 10(5):33–41, September 1990.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Siggraph 2000 Proceedings*, pages 335–342, 2000.

[RAA00] Constantino Carlos Reyes-Aldasoro and Ana Laura Aldeco. A combined algorithm for image segmentation using neural networks and 3d surface reconstruction using dynamic meshes. In *Rev Max Ing Biomed*, volume 21, pages 73–81, 2000.

[Rag93] I. Ragnemalm. The euclidean distance transform in arbitrary dimensions. *Pattern Recognition Letters*, 14:883–888, 1993.

[Ree83] W. T. Reeves. Particle systems–a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[Req80] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv.*, 12(4):437–464, 1980.

[Rey82] Craig W. Reynolds. Computer animation with scripts and actors. *SIGGRAPH Comput. Graph.*, 16(3):289–296, 1982.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM Press.

[RGR97] J. Rajapakse, J. Giedd, and J. Rapoport. Statistical approach to segmentation of single-channel cerebral mr images, 1997.

[RL00] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *SIGGRAPH 2000)*, pages 343–352, August 2000.

[Rod03] David Rodgman. *Refraction in Volume Graphics*. PhD thesis, Swansea University, 2003.

[Roe] Stefan Roettger. The volume library. http://www9.cs.fau.de / Persons / Roettger / library/.

[RP66] Azriel Rosenfeld and John L. Pfaltz. Sequential operations in digital picture processing. *J. ACM*, 13(4):471–494, 1966.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In Stephan N. Spencer, editor, *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 109–118, N. Y., 2000. ACM Press.

[RSSSG01] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast Volumetric Deformation on General Purpose Hardware. In *Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2001.

[RTSD03] Patrick Reuter, Ireneusz Tobor, Christophe Schlick, and Sebastien Dedieu. Point-based modelling and rendering using radial basis functions. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 111–118, New York, NY, USA, 2003. ACM Press.

[Rut99] Zsófia M. Ruttkay. Constraint-based facial animation. In *1024*, page 25. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-3681, 31 1999.

[RV06] D. Ruijters and A. Vilanova. Optimizing GPU volume rendering. In *WSCG - Winter School of Computer Graphics*, pages 9–16, 2006.

[RVG06] Peter Rautek, Ivan Viola, and Meister Eduard Gröller. Caricaturistic visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1085–1092, 2006.

[Sat01] Richard A. Satherley. *Computation and Applications of Distance Fields in Volume Graphics*. PhD thesis, Swansea University, 2001.

[SBM94] Clifford M. Stein, Barry G. Becker, and Nelson L. Max. Sorting and hardware assisted rendering for volume visualization. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 83–89, New York, NY, USA, 1994. ACM Press.

[SCD+03] Matús Straka, Alexandra La Cruz, Leonid Dimitrov, Milos Srámek, Dominik Fleischmann, and Meister Eduard Gröller. Bone segmentation in ct-angiography data using a probabilistic atlas. In *Vision, Modeling and Visualization*, pages 505–512. VMV, November 2003.

[Sch90a] Phillip J. Schneider. A bézier curve-based root-finder. In Andrew S. Glassner, editor, *Graphics Gems*, chapter 8, pages 408–415. Academic Press Professional, 1990.

[Sch90b] Phillip J. Schneider. Solving the nearest-point-on-curve problem. In Andrew S. Glassner, editor, *Graphics Gems*, chapter 11, pages 607–611. Academic Press Professional, 1990.

[Sea97] G. Sealy. Representing and rendering sweep objects using volume models. In *CGI '97*, pages 22–27, Washington, DC, USA, 1997.

[SH05] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. In Randima Fernando, editor, *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General Purpose Computation*, chapter 20, pages 313–329. Addison-Wesley, 2005.

[Sha81] Michael Shantz. Surface definition for branching, contour-defined objects. *SIGGRAPH Comput. Graph.*, 15(2):242–270, 1981.

[SHN03] Anthony Sherbondy, Mike Houston, and Sandy Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[SJ02] Richard Satherley and Mark W. Jones. Hypertexturing complex volume objects. *The Visual Computer*, 18(4):226–235, 2002.

[SK00] M. Sramek and A. E. Kaufman. *Volume Graphics*, chapter vxt : A class library for object voxelisation, pages 243–252. Springer, 2000.

[SP86] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. In *SIGGRAPH '86: Proceedings of the 13th annual con-*

ference on Computer graphics and interactive techniques, pages 151–160, New York, NY, USA, 1986. ACM Press.

[Spe] Ben Spencer. Igneus ray tracer. `http://www.igneus.co.uk`.

[SS91] Paul Gene Swann and Sudhanshu Kumar Semwal. Volume rendering of flow-visualization point data. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 25–32, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[SS04] Vikas Singh and Deborah Silver. Interactive volume manipulation with selective rendering for improved visualization. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics (VV'04)*, pages 95–102, Washington, DC, USA, 2004. IEEE Computer Society.

[SSC03] V. Singh, D. Silver, and N. Cornea. Real-time volume manipulation. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 45–52, 2003.

[SSKE05a] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware–based raycasting. In E. Gröller, I. Fujishiro, K. Mueller, and T. Ertl, editors, *Volume Graphics*, pages 187–195. Eurographics, 2005.

[SSKE05b] Simon Stegmaier, Magnus Strengert, Thomas Klein, and Thomas Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics*, pages 187–195, 2005.

[ST90] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization*, pages 63–70, New York, NY, USA, 1990. ACM Press.

[SWY99] N. Shareef, D.L. Wang, and R. Yagel. Segmentation of medical images using legion. 18(1):74–91, January 1999.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[THB$^+$90] Ulf Tiede, Karl Heinz Hoehne, Michael Bomans, Andreas Pommert, Martin Riemer, and Gunnar Wiebecke. Surface rendering. *IEEE Computer Graphics and Applications*, 10(2):41–53, 1990.

[TK95] H. Tek and B. B. Kimia. Shock-based reaction-diffusion bubbles for image segmentation. In Nicholas Ayache, editor, *Computer Vision, Virtual Reality and Robotics in Medicine*. Springer-Verlag, 1995.

[TPBF87] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214, New York, NY, USA, 1987. ACM Press.

[TPG99] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: improved iso-surface extraction. *Computers and Graphics*, 23(4):583–598, 1999.

[ult] Ultrasona of columbus ohio. `http://www.ultrasonacolumbus.com/`.

[VKG03] Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller. Hardware-based nonlinear filtering and segmentation using high-level shading languages. In K. Moorhead G. Turk, J. van Wijk, editor, *Proceedings of IEEE Visualization 2003*, pages 309–316. IEEE, October 2003.

[VKG04] Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller. Importance-driven volume rendering. In *Proceedings of IEEE Visualization'04*, pages 139–145, 2004.

[VS91] L Vincent and P Soille. Watersheds in digital spaces: An efficient algorithm based on immersion simulations. 13(6):583–598, 1991.

[vW84] Jarke J. van Wijk. Ray tracing objects defined by sweeping planar cubic splines. *ACM Trans. Graph.*, 3(3):223–237, 1984.

[WBMS01] Ross T. Whitaker, David E. Breen, Ken Museth, and Neha Soni. A framework for level set segmentation of volume datasets. In *Volume Graphics*, 2001.

[WC01] A.S. Winter and M. Chen. *vlib*: A volume graphics API. In *Volume Graphics 2001*. Springer-Wien New York, 2001.

[WC02] A.S. Winter and M. Chen. Image-swept volumes. *Computer Graphics Forum*, 21(3):441–441, 2002.

[WEE03] D. Weiskopf, K. Engel, and T. Ertl. Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.

[Wes89] Lee Westover. Interactive volume rendering. In *VVS '89: Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, New York, NY, USA, 1989. ACM Press.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. *CG*, 24(4):367–376, August 1990.

[Whi79] Turner Whitted. An improved illumination model for shaded display. In *SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14, New York, NY, USA, 1979. ACM Press.

[Wika] Computed tomography - wikipedia.

[Wikb] Magnetic resonance imaging - wikipedia.

[Wikc] Medical ultrasonography - wikipedia.

[Win02] Andrew S. Winter. *Field-Based Modelling and Rendering*. PhD thesis, Swansea University, 2002.

[Wit01] Craig M. Wittenbrink. R-buffer: a pointerless a-buffer hardware architecture. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 73–80, New York, NY, USA, 2001. ACM Press.

[WJ06] S.J. Walton and M.W Jones. Volume wires : A framework for empirical nonlinear deformation of volumetric datasets. *Journal of WSCG*, 13:81–88, 2006.

[WJ07] Simon Walton and Mark Jones. Interacting with volume data : Deformations using forward-projection. In *Fourth International Conference Medical Information Visualisation - BioMedical Visualisation (MediViz 2007)*, 2007.

[WK95] Sidney W. Wang and Arie E. Kaufman. Volume sculpting. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 151–ff., New York, NY, USA, 1995. ACM Press.

[WKE02] Manfred Weiler, Martin Kraus, and Thomas Ertl. Hardware-based view-independent cell projection. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 13–22, Piscataway, NJ, USA, 2002. IEEE Press.

[WL93] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(11):1101–1113, 1993.

[WMFC02] Brian Wylie, Kenneth Moreland, Lee Ann Fisk, and Patricia Crossno. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 7–12, Piscataway, NJ, USA, 2002. IEEE Press.

[WMLK89] J.W Wallis, T.R Miller, C.A Lerner, and E.C. Kleerup. Three-dimensional display in nuclear medicine and radiology. *Nuclear Medicine*, 32(3):534–546, 1989.

[WNDO99] Mason Woo, Jackie Neider, Tom Davis, and OpenGL Architecture Review Board. *OpenGL Programming Guide: the Official Guide to Learning OpenGL, version 1.2, Third Edition*. Addison-Wesley, Reading, MA, USA, 1999.

[Wol98] George Wolberg. Image morphing: a survey. *The Visual Computer*, 14(8/9):360–372, 1998.

[WP00] Zhongke Wu and Edmond C. Prakash. Volume graphics. chapter Visible Human Animation, pages 243–252. Springer, 2000.

[WPSH06] C. Wyman, S. Parker, P. Shirley, and C. Hansen. Interactive display of isosurfaces with global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):186–196, 2006.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88: Proceedings of the*

15th annual conference on Computer graphics and interactive techniques, pages 85–92, New York, NY, USA, 1988. ACM Press.

[WRS01] Rüdiger Westermann and Christof Rezk-Salama. Real-time volume deformations. *Computer Graphics Forum*, 20(3):443–451, 2001.

[WZMK05] Lujin Wang, Ye Zhao, Klaus Mueller, and Arie E. Kaufman. The magic volume lens: An interactive focus+context technique for volume rendering. In *IEEE Visualization*, page 47, 2005.

[XM07] Xianghua Xie and Majid Mirmehdi. Mac: Magnetostatic active contour model. *IEEE Transactions on Pattern Analysis and Machine Intelligence, to appear*, December 2007.

[YS93] Roni Yagel and Zhouhong Shi. Accelerating volume animation by space-leaping. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 62–69, 1993.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: An interactive system for point-based surface editing. In Stephen Spencer, editor, *SIGGRAPH 2002*, volume 21, 3 of *ACM Transactions on Graphics*, pages 322–329, New York, July 2002. ACM Press.

[ZPvBG01a] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Ewa volume splatting. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 29–36, Washington, DC, USA, 2001. IEEE Computer Society.

[ZPvBG01b] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH 2001*, pages 371–378, New York, NY, USA, 2001. ACM Press.

[ZRB$^+$04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004*, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

List of Figures

List of Tables

List of Algorithms

Listings