Advanced Techniques for the Visualization of 3D Medical Datasets

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Ángel del Río Fernández
aus Santiago de Compostela

Tübingen
2006

Tag der mündlichen Qualifikation:	*23.03.2006*
Dekan:	Prof. Dr. M. Diehl
1. Berichterstatter:	Prof. Dr.-Ing. Dr.-Ing. E.h. W. Straßer
2. Berichterstatter:	Daniela Tost Pardell (Profesor Titular, Doctor)
	Universitat Politècnica de Catalunya - Spanien
3. Berichterstatter:	Montserrat Bóo Cepeda (Profesor Titular, Doctor)
	Universidad de Santiago de Compostela - Spanien

Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die im Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben der Quellen als Entlehnung kenntlich gemacht worden sind.

Tübingen, Januar 2006 *Ángel del Río Fernández*

To my family

Zusammenfassung

Die Visualisierung zählt zu den ältesten und wichtigsten Anwendungen der Computergraphik. In den letzten Jahrzehnten hat die Bedeutung der Volumenvisualisierung exponentiell zugenommen. Dieses Wachstum war von besonders großer Bedeutung für die Anwendung der 3D-Visualisierung in der Medizin, wo die Verbreitung immer genauerer Scanner zur Entwicklung neuer Darstellungs-Verfahren für die grösseren und genaueren Datensätze geführt hat.

In dieser Doktorarbeit werden verschiedene Lösungsanšatze für die korrekte Visualisierung von 3D-Daten vorgestellt, wobei der Anwendungsschwerpunkt immer auf dem medizinischen Bereich liegt. Aufgrund der Vielfalt existierender Visualisierungsalgorithmen, und um einen korrekten, umfassenden Überblick zu schaffen, ohne auf die relevanten Details zu verzichten, besteht diese Arbeit aus drei unterschiedlichen thematischen Einheiten:

Die erste Einheit, die in Kapitel 2 präsentiert wird, beschäftigt sich mit *punktbasierten Darstellungstechniken (Point Rendering)* und ihre Anwendung auf die graphische Darstellung von volumetrischen Daten. Die inhärente Einfachheit der Verwendung von Punkten als Darstellungs- und Modellierungsprimitive, sowie die wachsende Leistung von Graphikhardware, machen *punktbasiertes Rendering* zu einer interessanten Alternative zu traditionellen Darstellungstechniken. Trotzdem stellt das Erreichen von Echtzeit- oder selbst interaktiven Bildfrequenzen angesichts der grossen Menge von Punktprimitiven, die für die Darstellung grosser 3D-Modelle notwendig sind, eine Herausforderung dar. Unsere Arbeit in diesem Bereich befasst sich mit einer hardwarebasierten Lösung, die eine hohe Geschwindigkeit erreicht, oh-

ne die Bildqualität zu verringern. Um die Ausführungen verständlicher und vollständiger zu machen, wird zuerst ein Überblick über die wichtigsten Algorithmen für punktbasierte Darstellung gegeben. Nachdem die grundlegende Funktionsweise erklärt wurde, wird unsere hardwarebeschleunigte Punktprojektionstrategie erläutert und im Detail analysiert.

In der zweiten Einheit (Kapitel 3) liegt der Schwerpunkt auf den Möglichkeiten der *indirekten Volumendarstellung (indirect Volume Rendering)* als Volumenvisualisierungswerkzeug. In der indirekten Volumendarstellung werden charakteristische Isoflächen extrahiert und anschließend dargestellt, um einen Einblick im Volumendatensatz zu gewinnen. Da die Isoflächenextraktion das zentrale Element in jeder Anwendung der indirekten Volumendarstellung ist, wird die Standardmethode, d.h. der *Marching Cubes* Algorithmus, hier im Detail erklärt. Das notwendige Hintergrundwissen wird mit einer Analyse der wichtigsten Beschleunigungstechniken ergänzt, so dass der aktuelle Stand der Technik aufgezeigt wird. Auf diesem aufbauend wird dann unsere eigene Arbeit vorgestellt, die auf eine hardwarebeschleunigte Isoflächendekodierung und -darstellung abzielt.

Kapitel 4 schliesst den Kreis von Volumenvisualisierungsalgorithmen mit einer Einheit über *direkte Volumendarstellung (direct Volume Rendering)*. Als die unmittelbare Darstellung eines 3D-Datensatzes ist *direct Volume Rendering* der ideale Ansatz, um einen guten Einblick in einen kompletten Patientenscan auf einmal zu erhalten. Allerdings ist die Erzeugung einer klaren Darstellung mit einer Betonung von signifikanten Bereichen gegenüber von Kontextinformation keineswegs trivial, da die Auflösung der Aufnahmegeräte (z.B. CT-/MRI-Scanner) ständig wächst, und mit ihr auch die entsprechenden Datensätze. Darüberhinaus ist die Anwendung der Techniken, die für die Betonung dieser Regionen zuständig sind, oft zu kompliziert, was einen praktischen Einsatz im medizinischen Bereich meist verhindert. Deswegen fangen wir in dieser Einheit mit einem Überblick über die Grundlagen von *direct Volume Rendering* und einer kurzen Sammlung der bekanntesten Algorithmen an, um uns dann mit unserem eigenen Ansatz für die Volumenklassifikation zu befassen. Unsere Methode kombiniert eine intuitive *Augmented Reality*-Umgebung zur direkten Darstellung und Benutzerinteraktion mit einem automatischen Klassifizierungsverfahren, das auf maschinellen Lernverfahren basiert. Auf diese Art und Weise wird dem Arzt ein komplettes Werkzeug zur Verfügung gestellt, mit dem wichtige Materialien im Volumen halbautomatisch hervorgehoben werden können, ohne dass man ein Experte in der Definition von Transferfunktionen sein muss.

Abschließend werden verschiedene Beispiele von praktischen Anwendungen, die zu konkreten Projekten gehören, in Appendix A zusammengefasst. Dadurch wird die Einsatzfähigkeit der medizinischen Volumenvisualisierung in realen Anwendungen illustriert.

Abstract

Visualization is probably one of the oldest and most important applications of computer graphics. In the last decades the relevance of volume visualization has increased exponentially. This growth has been especially significative in the application of 3D visualization to medicine, where the development of more accurate scanning devices has escorted the creation of new algorithms to enable the proper display of these larger and more precise data.

This thesis presents several solutions for the correct visualization of 3D data, emphasizing their utility as a tool in the medical field. Due to the diversity of volume visualization algorithms, and in order to provide a correct global overview without sacrificing necessary and relevant details, this work has been structured in three different thematic units:

The first unit, presented in Chapter 2, is devoted to *point rendering* and its application for obtaining graphical representations of volumetric data. The simplicity inherent to the utilization of point samples as rendering and modelling primitives and the increasing performance of graphics hardware have made point rendering an interesting alternative to more traditional rendering modalities. Nevertheless, given the large amount of point primitives necessary to represent large 3D models, real-time or even interactive framerates can become a challenge. Our work in this field is devoted towards a hardware-based solution that enables high speed point-based rendering without sacrificing image quality. In order to make the exposition fully understandable and self-contained, a survey of the most important algorithms for point-based rendering is firstly exposed. Once the working principle of these methods has been presented, our hardware accelerated point projection

strategy is explained and analyzed in detail.

In the second unit, enclosed in Chapter 3, the focus is shifted to the possibilities of *indirect volume rendering* as a volume visualization tool. In indirect volume rendering, representative isosurfaces are extracted and rendered as a means to gain an insight of the volume data. Since isosurface extraction is a key element in every indirect volume rendering application, the standard technique for this purpose, i.e., the *marching cubes* algorithm is explained in detail here. The necessary background is complemented with an analysis of the most relevant acceleration techniques, hence indicating the current state-of-the-art on the field. This is then further extended with the presentation of our own work, which pursues a hardware (GPU) accelerated isosurface decoding and rendering solution.

Chapter 4 closes this circle of volume visualization algorithms with a *direct volume rendering* unit. As a straight representation of a 3D dataset as a whole, direct volume rendering is the perfect option to obtain a good insight of a patient's scan at once. However, given the increasing resolution of the acquisition devices (e.g., CT-/MR-scanners) and therefore of the generated datasets, the production of a clear representation, where certain regions of interest are highlighted over other context information, is by no means a trivial task. Furthermore, the internal complexity of the techniques responsible of emphasizing these regions of interest too often prevents new interesting algorithms from being applied in practical cases. Therefore, in this unit we start by first giving an overview of the principles of direct volume rendering together with a brief collection of the most commonly used algorithms, for then focussing on our novel approach for volume classification. Our method combines an intuitive *augmented reality* environment for direct display and user interaction, with automatic machine-learning based classifiers. This way, the physician is furnished with a complete tool able to efficiently highlight materials and features of interest within the volume in a semiautomatic way, without requiring from her/him a high level of expertise on transfer function design.

Finally, several examples of practical applications corresponding to concrete projects are summed up in Appendix A, thus illustrating the applicability of medical volume visualization to real-life cases.

Acknowledgements

First, I would like to thank very specially my advisor Prof. Wolfgang Straßer for giving me the opportunity to join his group WSI/GRIS and his continuous support and encouragement during these years in Tübingen. I also would like to thank Dr. Daniela Tost Pardell and Dr. Montserrat Bóo Cepeda who agreed to be part of my graduation committee.

Furthermore I would like to thank the members of the Visual Computing for Medicine group, Jan Fischer and PD Dr. Dirk Bartz without whom this work would probably not have been possible. Moreover I wish to thank all my colleagues at WSI/GRIS for their warm welcome and countless unforgettable moments. I also want to acknowledge the students with whom I had the pleasure to work with, especially Martin Köbele and Matthias Pfeifle, who always showed great disposition and dedication during their diploma and student theses and committed valuable contributions to the implementation of some of the presented projects.

Finally, I want to express my gratitude to my family and friends for their unconditional support throughout the years.

CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Visualization of 3D Medical Datasets

The visualization of scientific data is one of the main challenges in computer graphics. Since its recognition in 1987 in the NSF report [88] as a field on its own, scientific visualization has grown and developed until a point where it has become indispensable to interpret and understand all kinds of results obtained in many different fields. From the simulation of fluids dynamics, to the analysis of multidimensional hypersurfaces, or the inspection of scanned data, the use of computed visualization techniques has played a crucial role in the development of modern science and its applications.

In this thesis, we focus on the application of visualization techniques to the medical activity. In medicine, most data employed for diagnosis, intraoperative assistance, postoperative analysis or even for teaching purposes, such as radiographs, computed tomography (CT) scans, magnetic resonances (MRI), positron emission tomography (PET) images, etc., are obtained using scanning devices. These devices usually sample a continuous magnitude of the patient's anatomy, e.g., Hydrogen density, thus yielding to a discrete representation. While traditional acquisition techniques work on a 2D basis, the implantation of 3D scanners has grown immensely during the last decades, hence bringing much attention from the medical imaging community into 3D computer graphics and scientific visualization methods.

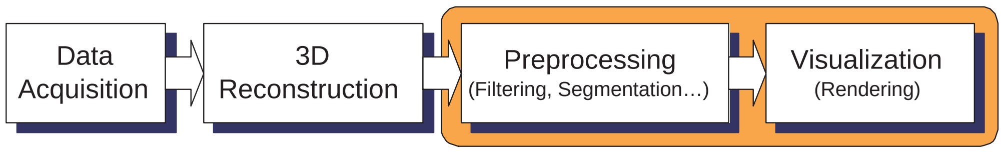


Figure 1.1: General workflow in medical visualization.

The general structure of any medical visualization application responds to the workflow depicted in Figure 1.1. Initially, the patient is examined by means of a scanning device. These scanning apparatus measure and sample a given magnitude representative of the patient's anatomy along a certain direction. All these acquisition tools stem from the pioneer work done by Wilhelm Conrad Röntgen (1845-1923), who in 1895 accidentally discovered a new type of radiation which was later known as X-rays. The utilization of X-rays in medical practice was almost immediate and has continuously been used, through a large path of improvements, until today. The basic principle of measuring the interaction of an external source of energy and the human body matter is still the core of all scanning devices utilized in today's medical imaging. This spectrum includes from high energy electromagnetic radiation in X-rays and computed tomography (CT) to an external magnetic field in magnetic resonance imaging (MRI). But of course, contemporary scanning equipment has little to do with the experimental tubes designed by Röntgen over one century ago. In present day's routine, most systems are digital and radiation doses (X-ray, CT) have been severely reduced, while the resolution has tremendously increased. Even more, the development of more complex emitters and detectors, as well as more powerful and accurate magnetic field generators has lead, additionally to better images, to a wider spatiotemporal range of acquisition yielding to 3D and even 4D datasets.

A common characteristic of all these devices is the necessity of processing the measured data in order to reconstruct the spatial structure of the patient. This process, called reconstruction is based on back-projection techniques. Radon and Fourier reconstruction methods are among the most commonly applied to this purpose. In this work, the emphasis is put on 3D visualization, meaning that the result of this reconstruction phase is a volumetric dataset usually comprising a set of parallel slices. Such volumes constitute the basis for the visualization techniques that are discussed in this thesis.

Additionally, as the diagram in Figure 1.1 illustrates, an intermediate step might be necessary between the reconstruction of the 3D dataset and its final presentation to the user. This element, labelled in the figure as *preprocessing* in general, can comprise many different transformations and data analysis algorithms. Typical examples are the segmentation of structures of interest (e.g., bone structures or blood vessels), the application of filters to reduce the presence of noise in the dataset, to prepare the data for further processing or to extract additional information and provide a better understanding of the scanned body (e.g., edge detection, gradient estimation), etc. The limit between preprocessing for preparing the data to be visualized and the actual application of a visualization algorithm is often only theoretical and these two stages of the pipeline can then be easily merged into a single one. The focus in this thesis is to present and analyze a set of alternatives for addressing the difficulties raised by these two last elements of the workflow of a medical visualization application, i.e., algorithms for the visualization of 3D medical datasets and, when necessary, the corresponding preprocessing associated to them. Taking into account this framework, let us pose some of the main challenges to be confronted in today's 3D medical visualization.

The steady technical improvement experimented in the last decades by acquisition devices and reconstruction algorithms has produced also a noticeable refinement in the temporal and spatial resolution of the generated 3D datasets. While this evolution has multiplied the diagnosis and treatment efficiency in many cases, it also means new challenges from a computing point of view, as the size of the data has also exploded. Therefore, new algorithms must be and are being designed in order to cope with the new difficulties these large datasets imply. Most novel visualization and rendering techniques can be classified into two complementary groups according to their intended target: on the one hand, those dedicated to **accelerate** the display of this large amount of data and, on the other hand, those techniques devoted to find efficient ways to gain a **better insight** by highlighting the most *relevant* parts of the dataset.

As both groups are of utter importance for the development and improvement of medical and scientific visualization, algorithms belonging to both categories are presented in this thesis. But before providing an overview of these techniques and the way they have brought forward the current state-of-the-art in Section 1.2, a brief survey is presented of different criteria for the classification of visualization solutions. By observing the group a given algorithm belongs to and the common characteristics shared by all methods of a kind, a better and more complete perception of every specific technique can be easily achieved. Therefore, a comprehensive categorization is of big interest for a thorough understanding of each algorithm and the recognition of its added value and originality.

A basic criterion for the classification of visualization methods is the rendering strategy they make use of. One crucial aspect of this rendering algorithm's classification is the type of primitive employed to represent the structures to be visualized. While traditionally in computer graphics the standard modelling and rendering primitives have always been polygons, and predominantly triangles, other possibilities have also been explored and successfully applied. This way, according to the primitive they use, rendering algorithms can be included in one of the following three categories: *polygon-based rendering*, *image-based rendering* and *point-based rendering*.

In polygon-based rendering, the elements of the scene to be displayed are modelled as a set of polygonal meshes, usually composed of triangles. The number and size of the triangles determines the resolution of the model, a factor that plays a crucial role in the quality of the final image. In volume visualization applied to medical purposes, polygon-based rendering is very often utilized for displaying surfaces representative of relevant inner structures of the volume. These surface models can be generated as the result of segmentation, the extraction of an isosurface, etc.

Due to the large number of triangles necessary to represent highly detailed models, alternatives to polygon-based rendering, such as image-based and point-based rendering, have been investigated. In image-based rendering algorithms, the appearance of three dimensional models is represented by 2D images of the object. However, a purely image-based representation is too expensive in terms of memory requirements, if it is to be realistic, mainly due to the high dimensionality of the problem (5D). Therefore, hybrid geometry and image-based approaches and simplifications are usually utilized instead. Clear examples of this category are image-based impostors [114] (e.g., billboards [107]), or texture mapping [32, 55].

In texture mapping, images are mapped onto parameterized surfaces to provide additional details supplanting high resolution geometry. There exist many variants of texture mapping, but specially relevant for volume visualization are the three ways in which texture mapping may be used in volume rendering, i.e., to obtain an image of a solid, translucent object. The first is to draw slices of the object from back to front [43]. Each slice is drawn by first generating a texture image of the slice by sampling the data representing the volume along the plane of the slice, and then drawing a texture mapped polygon to produce the slice. Each slice is blended with the previously drawn slices using transparency. The second method uses 3D texture mapping [42]. In this method, the volumetric data is copied into the 3D texture image.

Then, slices perpendicular to the viewer are drawn. Each slice is again a texture mapped polygon, but this time the texture coordinates at the polygon's vertices determine a slice through the 3D texture image. This method requires a 3D texture mapping capability, but has the advantage that texture memory needs to be loaded only once independently of the viewpoint. A third way is to use texture mapping to implement *splatting* as described by [131, 72].

Point-based rendering uses sample points from surface models as display primitives. This has the advantage of not having to deal with any explicit representation of neighborhood relationships, as with polygon-based rendering. By doing so, not only are the memory storing requirements reduced, but it also facilitates the application of reduction and simplification techniques to render complex scenes. Due to these and other properties, together with the development of new graphics hardware, point-based rendering has become a practical alternative to polygon-based rendering for modelling and visualizing large and complex scenes.

At last, another common classification for rendering algorithms splits them into *forward mapping* and *backward mapping* techniques, depending on the way the visibility of elements in a scene is solved. Forward mapping algorithms operate in world space and typically sort all objects to be rendered with the assistance of a z-buffer [32, 119] so that background objects be occluded by foreground ones. Backward mapping tackles the visibility problem in the opposite direction. Instead of processing the primitives and sorting them according to their depth values, with backward mapping a search is performed among the elements in the scene for each pixel in the image to determine which objects are visible and should be displayed. A typical example of backward mapping is the ray-tracing algorithm [22, 132] and its simplification for volume rendering, ray-casting [74].

1.2 Overview and Contributions

As mentioned above, in this thesis, several methods are described that focus on both reducing the time required for rendering a dataset, and improving the quality or meaningfulness of the produced images. More specifically, novel performance-driven algorithms in the field of point rendering and indirect volume rendering are presented, as well as a new quality-driven technique for direct volume rendering. As a whole, a survey of the three rendering strategies oriented to their application to medical 3D visualization is globally depicted. Special emphasis is put on our own contributions and the way they have extended the current state-of-the-art in every case.

Each chapter starts with a brief introduction which summarizes the most relevant related work already existing in the field, as well as background information necessary to properly understand the new proposed algorithm. A detailed explanation and analysis of each novel technique is then performed and complemented with illustrative examples.

First, in Chapter 2, the characteristics of point-based rendering as an alternative to traditional polygon-based rendering for the visualization of large and complex scenes is presented. After reporting the necessary basics on the functioning of point rendering algorithms, our work on high-speed projection for point rendering applications is introduced. This work, that establishes the basis for a high performance and fully hardware-accelerated solution for point rendering, was originally published in [1].

Chapter 3 focuses on indirect volume rendering and its application to the visualization of 3D medical datasets. As isosurface extraction is a decisive element of indirect volume rendering, the most commonly used technique (i.e., the *Marching Cubes* algorithm) and a survey of several existing optimizations are reported. Next, the problem of rendering large extracted isosurfaces is tackled, concentrating on our GPU-based proposal presented in [6], for decoding and rendering compressed isosurfaces directly on the graphics card. This was, at the time of first publication, the first application where compressed isosurfaces were both decompressed and rendered fully on the GPU, thus avoiding any slow read-back to the CPU and speeding-up rendering.

After the presentation of performance-oriented techniques in the two previous chapters, the algorithms discussed in Chapter 4 address the question of how to improve the quality and value of the visualization experience in a medical scenario. Initially, a quick reminder is provided of some basic concepts on direct volume rendering, such as the functional pipeline or the most common rendering algorithms. Then the focus is shifted to the classification stage, where all optical properties (e.g., color, opacity) for rendering are assigned. In this particular topic, our novel approach originally presented in [7, 8] is described in detail. This half-automatic solution combines modern classification techniques, such as the use of 2D histograms, with machine learning methods (e.g., neural networks) in order to assist the user during the definition of an appropriate transfer function. By utilizing an *Augmented Reality (AR)* environment for the user interaction and display of the results, a highly intuitive visualization experience is achieved.

Finally, in Appendix A, a survey of research projects is presented, which show real-life cases where some of the here described algorithms and techniques were utilized for practical purposes in medical volume visualization applications.

CHAPTER 2

Point Rendering

2.1 Introduction

Point-based rendering constitutes an alternative to the classical rendering approaches: *polygon-based* and *image-based rendering.* The use of polygons, normally triangles, is the standard avenue in most computer graphics applications for representing the different objects composing a scene. Each object is modelled as a set of connected vertices forming a mesh that reproduces its outermost surface. While this can yield to highly appealing rendering results and good rendering performance, its major drawback is the necessity of storing explicit connectivity information together with the vertex positions. In complex scenes this can become a bottleneck that prevents real time or even interactive visualization.

Image-based techniques try to overcome this limitation by replacing complex objects with image-based impostors [114], such as billboards [107]. When using impostors, the actual geometry of the object is substituted by a previously generated image, therefore saving storage and processing resources, thus speeding up rendering. However, these image-based replacements produce severe parallax issues, which may lead to visible and disturbing artifacts in interactive scenarios. In order to overcome such limitations, many different images must be employed to achieve an acceptable representation of the object and hence considerably increasing the storage and processing requirements.

The use of sample points as rendering primitives presents several advantages compared to the both aforementioned classical approaches. In point-based rendering, each object in a scene is represented by a dense set of points lying on its surface. In contrast to polygon-based representations (typically triangle meshes), by using points as modelling primitive no explicit connectivity information must be stored. This does not only imply a reduction in terms of storage requirements, but it also facilitates the application of simplification algorithms to complex objects. Since in point rendering the view of an object is constructed from view-independent surface points in object space, no parallax problems arise as with image-based approaches.

2.2 Point Rendering Algorithms

Points have often been used for rendering purposes. However, even though points have been employed with particle systems to model smoke, clouds, dust, fire, water and trees [102, 117, 103], and to model solid objects [37], it has not been until recent years that point-based rendering has been accepted as a valid and promising alternative to classical rendering primitives. The basis for this development was first settled by Levoy and Whitted in 1985 for the case of continuous, differentiable surfaces [76], where a parametric smooth surface is converted into a point-based representation for further processing and rendering. The solutions proposed by Levoy and Whitted for rendering point sample models have been further extended and utilized in some recent proposals [138, 130].

Point-based techniques have also been used for 3D modelling purposes. In [121], each object is modelled using elastic surfaces defined as a set of *oriented particles*, which the authors call *surfels*. Attraction and repulsion forces are used to distribute the particles over the surface favoring locally planar arrangements. Editing is enabled by adding or removing particles, as well as by modifying the interaction potentials responsible for the particle distribution. This has the advantage of allowing interactive modelling without the necessity of dealing with base meshes or boundary problems. Many other applications of point-based rendering have been proposed during the last two decades, covering areas like sampling and rendering of implicit surfaces [39, 134], or rendering of complex terrain models [49].

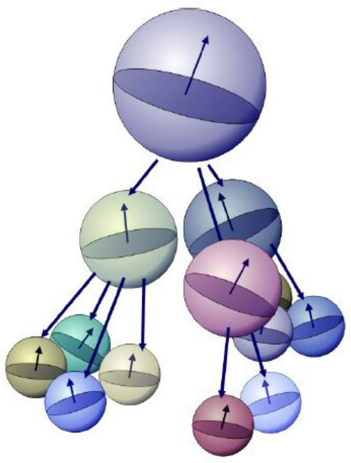


Figure 2.1: QSplat: Bounding sphere hierarchy with average attributes in each node [129].

Despite this considerable amount of existing research, it has not been until the last lustrum that point-based rendering techniques have become an actively utilized alternative to polygon-based and image-based techniques. Much responsibility for this sudden development is derived from the incorporation of multi-resolution strategies to point-based rendering. Point-based rendering can be more efficient than traditional rendering methods for complex models if triangles occupy a small screen area. This way, it is important to count with a level-of-detail (LOD) representation, able to identify the better suited rendering modality. Multi-resolution point-based rendering was introduced independently in 2000 by three different groups, which are now briefly presented.

QSplat

Rusinkiewicz and Levoy [108] proposed QSplat, a point-based rendering system relying on a hierarchy of bounding spheres for visibility culling, level-of-detail control and rendering. This approach was developed in order to render large amounts of scanned data within the *Digital Michelangelo* project [75]. A scanned triangular mesh is preprocessed and converted into a hierarchy of bounding spheres, where each node stores average surface attributes such as color and normal (see Figure 2.1). A quantization scheme is applied for a memory efficient encoding of the point hierarchy. During rendering, this hierarchy is traversed until the projected size of a sphere falls below a given

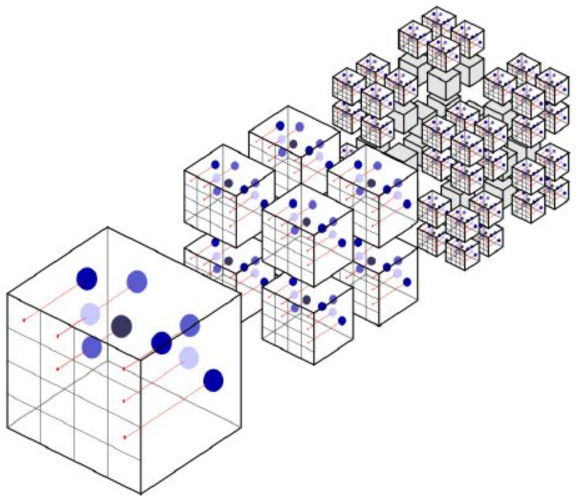
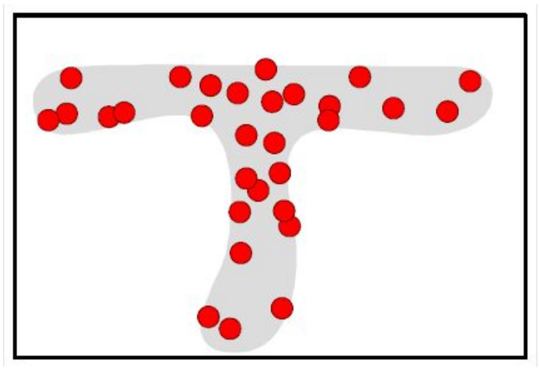


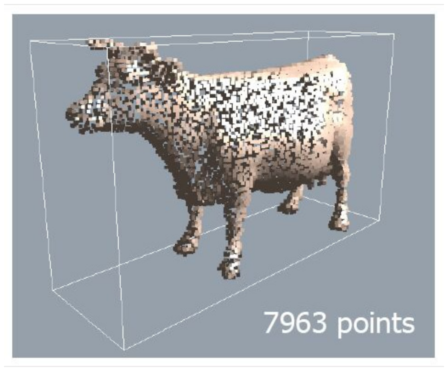
Figure 2.2: Surfels: Hierarchy of layered depth cubes. [129].

threshold for the splat size. Subsequently, fixed size splats with a color obtained as the average of the underlying points, are drawn into a z-buffer in order to obtain a complete representation avoiding holes in the final image.

Surfels

Also in 2000, Pfister et al. [99] presented a point-based rendering paradigm. In this case, the authors utilize oriented surface elements with attributes such as normals or prefiltered textures as rendering primitives – the so-called *surfels*. A sampling preprocessing step is first performed, in which textured geometrical objects in the scene are converted to a set of surfels. This is achieved by means of ray casting the scene along three orthogonal directions and storing all intersection points, thus creating three orthogonal layered depth images (LDIs) as in [113]. This set of LDIs, also called *layered depth cubes* (LDCs) [77], is organized in a hierarchical data structure forming an octree which can be accessed for rendering (see Figure 2.2). Each node in the octree represents one LDC with a sampling space dependent of its height in the octree. So the highest resolution is stored at the lowest level ($n = 0$), while the pixel spacing at a level n is $h_n = h_0 2^n$, being h_0 the spacing at the highest resolution. This leads to a multi-resolution representation where each child node contains geometry with twice as much accuracy. The rendering algorithm traverses the LDC tree top to bottom until the maximum distance between adjacent surfels in image space falls below a certain thresh-


(a) Random sampling



(b) Point samples

Figure 2.3: Randomized z-buffer [129].

old, which determines the degree of oversampling. The points corresponding to the selected level of detail are written into a z-buffer. For each surfel tangent disk, a parallelogram approximation of its projection is rasterized in order to detect holes in the z-buffer that would lead to visibility artifacts. Holes in the final image are then corrected during image reconstruction by applying either simple splatting with nearest neighbor interpolation, hierarchical push-pull [51], or gaussian interpolation with supersampling, leading to a tradeoff between image quality and rendering performance. Finally, for shading, linear interpolation between two adjacent levels in the hierarchy is employed to avoid popping artifacts.

Randomized Z-Buffer

The randomized z-buffer proposed by Wand et al. [130, 129] is also composed by three main stages: First, in a preprocessing step, a hierarchical representation of the scene to be rendered is constructed; then, during rendering, a traversal of the hierarchy is performed and point samples are generated; finally, the result is completed in an image reconstruction process. The main contribution of this technique compared to the two mentioned above, relies on the random nature of the sample points generation. This algorithm is based on the fact that, with an appropriate oversampling, a scene can be completely reconstructed from a set of randomly placed sample points on the surface of its objects (see Figure 2.3). In order to achieve a reasonable and sufficient point distribution, the projected area on the screen of each triangle should be employed. Since this would turn into too costly computations, an estimation of the projected area values is utilized instead. This approximation is calculated as a *projection factor*, a function of both the triangle's depth in the scene and its orientation. All triangles in the scene are classified into groups of similar projection factor. By using an octree, a spatial classification of regions within the scene is performed, thus bounding the depth factor within a node. Each node stores additionally a list of the summed area corresponding to the geometry it contains. During rendering, the octree is traversed in a view-dependent way and the geometry contained in the selected nodes is sampled. The selection of sample points is guided by a probability density function describing a uniform distribution on the projections of the surfaces on the image plane. This way, larger triangles on the screen are sampled more densely than those with smaller visual contribution to the final image. With a sufficient oversampling factor, it is guaranteed that at least one point is projected onto one pixel on the screen in the final image. During image reconstruction, a per-pixel reconstruction algorithm can be applied, where all sample points are taken in arbitrary order, projected onto the image plane and drawn as pixels using a z-buffer to resolve occlusions. Alternatively to per-pixel reconstruction, also splatting can be used to reduce rendering time and increase interactivity. Due to the considerable oversampling inherent to this algorithm, the costs associated to large triangles on the screen can easily overwhelm those of a conventional z-buffer. In order to avoid a drop in the rendering performance, triangles resulting in a large projected area on the screen are identified and rendered using a conventional z-buffer. The main advantage of the randomized z-buffer is that it can render the scene with arbitrary precision, not limited by the initial sampling like in the Surfels or QSplat approaches. On the other hand, the dynamic sampling applied in the randomized z-buffer technique can be more computationally expensive than using precomputed sample sets.

2.3 Hardware-Accelerated Point Rendering

One important drawback of all these point-based rendering algorithms is that, despite their proven suitability for rendering highly complex scenes, the existing graphics hardware is optimized for accelerating the rendering of triangles and not necessarily of points. Therefore, in the last years, several proposals have been presented for either trying to exploit the capabilities of current graphics cards, or developing alternative architectures more appropriate for dealing with the problems associated to point-based methods. In both cases, two complementary issues must be addressed for a successful result: the visual quality of the rendered images, and the rendering speed.

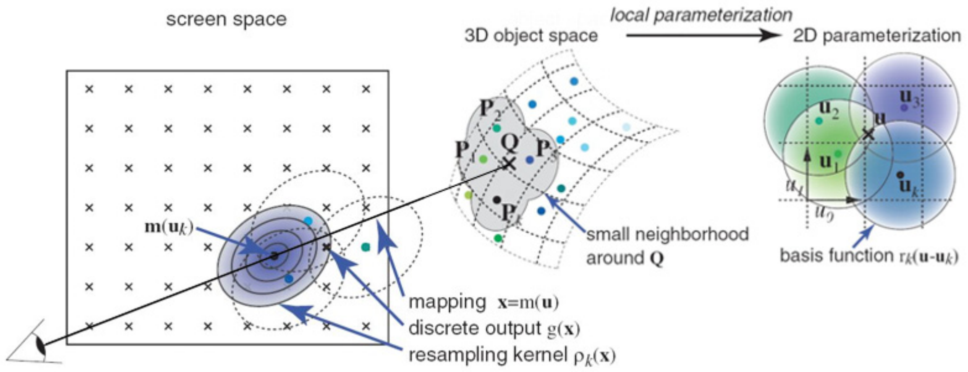

screen space
local parameterization
3D object space
2D parameterization
$\mathbf{m}(\mathbf{u}_k)$
$\mathbf{P}_2$
$\mathbf{P}_1$
$\mathbf{Q}$
$\mathbf{P}_3$
$\mathbf{P}_k$
$\mathbf{u}_2$
$\mathbf{u}_3$
$\mathbf{u}$
$\mathbf{u}_1$
$\mathbf{u}_k$
u_1
u_0
small neighborhood around $\mathbf{Q}$
basis function $r_k(\mathbf{u}-\mathbf{u}_k)$
mapping $\mathbf{x}=\mathrm{m}(\mathbf{u})$
discrete output $g(\mathbf{x})$
resampling kernel $\rho_k(\mathbf{x})$

Figure 2.4: EWA surface splatting [138].

Both aspects have to be efficiently solved in order to achieve a satisfactory hardware-accelerated point-based rendering strategy.

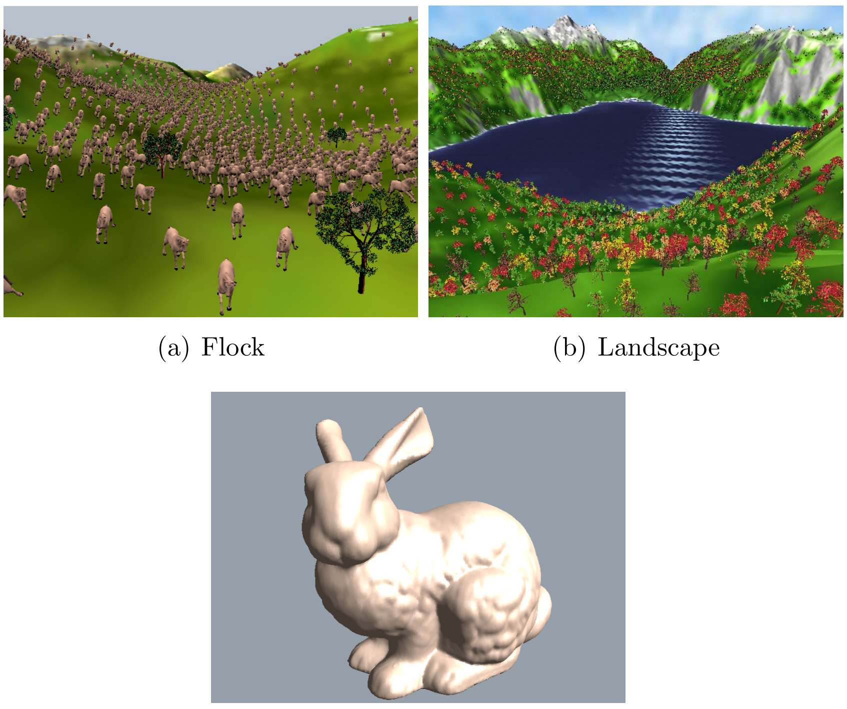
The visual quality resulting of point-based rendering algorithms is highly dependent on the way the point samples are combined to determine the color of each pixel. Since most algorithms rely on different types of splatting, hardware acceleration of this filtering process is of big relevance for the appearance of the final image. Based on the work of Zwicker et al. [138] on elliptical weighted average (EWA) surface splatting, Ren et al. [105] presented a hardware-accelerated approach that applies high quality, anisotropic texture filtering to complex point-based scenes. The original work of Zwickler et al. [138] performs screen space EWA splatting to reconstruct the final image out of a set of irregularly spaced textured points in three dimensional object space. Each point is associated with a radially symmetric basis function (elliptical Gaussian) on a locally parameterized domain indicating its color. These basis functions, which act as reconstruction filters, provide a continuous texture function on the surface represented by the set of points (see Figure 2.4). During rendering, this continuous texture function is warped to screen space using a local affine mapping of the perspective projection at each point, and the result is sampled at each screen space coordinate in order to produce the final discrete image. This way, the output of the algorithm - each pixel color - can be obtained as a weighted sum of screen space resampling filters.

The configuration described above is suitable for software implementations, with the limitations that this implies. A better rendering performance can be achieved with a hardware-accelerated approach. In [105], Ren et al.

propose the conversion of the screen space EWA splatting into an *object space EWA splatting*. In the latter case, the resampling filter (a Gaussian again) is formulated as a function on a parameterized surface in object space, and then, the result is projected to screen space, yielding the rendered image. Modern GPUs programmability (vertex and fragment shaders) is used to implement the filtering in a two-pass approach where, in the first pass, visibility splatting [99] is performed, and in the second pass the actual filtering and mapping are produced. This way, antialiased images are obtained at higher rates than with a software implementation, without loss of quality.

An alternative approach for hardware acceleration, also based on the surfels algorithm of Pfister et al. [99], was proposed by Coconu and Hege in [36]. This is, however, not a purely point-based rendering solution, but a hybrid system which combines the use of points and triangles in a hierarchical LOD representation in a similar way as in [130]. Here again, an off-line preprocessing step is required, where the LOD hierarchy for points and triangles is generated. Analogously to the work of Pfister et al. [99], also view independent EWA texture pre-filtering with Gaussian kernels and several mipmaps are employed to deal with minification problems. The main contribution of the work of Coconu and Hege [36] consists in the image reconstruction algorithm. They use what they call *fuzzy splats* of Gaussian elliptical form, which are suitable for being rendered using current graphics hardware features (point sprites, programmable geometry and texture pipelines). The selected points are ordered back to front and combined using fuzzy splats using a Gaussian distribution as transparency pattern for each splat. The degree of oversampling is determined by traversing the hierarchy structure during rendering according to a given threshold for image plane spacing between adjacent projected points. Then, the splats are combined using alpha-blending to produce the final image. Popping artifacts due to LOD transitions are avoided by utilizing alpha-blending between both the old (e.g. triangles) and the new representation (e.g. points).

Another proposal focussed on accelerating the rendering of point-sampled geometry without sacrificing visual quality is the work of Botsch and Kobbelt [30]. They also implement a GPU-based splatting algorithm with filtering and blending of several overlapping splats. The appearance of holes in the image is avoided by selecting the size and shape of the splats according to the current viewing parameters and the splat position in the scene. For splat filtering, each splat in object-space is associated a radially symmetric Gaussian weighting function, yielding elliptical Gaussian filters in image-space. The main contribution of this work consists on the fact that the whole splatting processing takes place on the GPU, thus freeing the CPU and profiting from


(a) Flock (b) Landscape

(c) Bunny

Figure 2.5: Test scenes obtained with point rendering [1].

the highly parallel architecture of vertex and – specially – fragment units.

2.4 High Speed Projection in Point Rendering Applications

All the proposals summarized above, try to accelerate the final stages of the rendering pipeline by adding filtering, visibility sorting, etc. in order to achieve appealing images in a point-based scenario. However, depending on the complexity of the scene and on the point-based rendering approach in use, other problems may arise as well. In our case, we concentrate in the randomized z-buffer algorithm [130, 129]. As explained previously, the randomized z-buffer relies on a sufficient oversampling of the scene, so that the maximum distance in the image plane between two adjacent points in object-space is of, at most, one pixel. This implies, for highly complex models, dealing with several millions of points in real-time (see Figure 2.5). Attempts to increase the speed of the system, through the utilization of hardware-accelerated approaches, permit the setup of a primitive every five clock cycles with typical clocking in the neighborhood of 300 MHz [84]. However, highly complex scenes with several millions of points require a higher order performance, reaching rates of up to one billion points per second [130]. This suggests the proposal of new hardware-accelerated approaches for the processing and management of large numbers of points. In this sense, the first algorithm involved in point rendering applications is the projection of points from object space to image space. Therefore, our work [1] concentrates on improving the performance of perspective projection on current graphics hardware, when applied to point-based rendering.

Current graphics cards are optimized for processing triangular meshes, where only the vertices of the mesh are projected, thus making the projection operation a non critical one. In point rendering applications, however, the increment in the number of points that have to be handled makes necessary a reconsideration of the classical projection strategy because of its computational core being based on the division operation and the dividers implemented in current graphics cards having a recursive structure. This requires the proposal and analysis of alternative algorithms for fast perspective projection, avoiding or minimizing divisions.

In the remainder of this chapter, we describe a new projection technique for perspective correct rendering that minimizes division operations. This technique is based on the identification of a projection area on the screen and the selection of the final pixels through comparison operations. The simplicity of this novel projection algorithm permits the employment of parallelization as a way to increase the processing speed. Additionally, an efficient scheduling strategy allows the usage of a pipelined structure with the consequent increase in the obtained performance. Following, a brief introduction to the classical projection and the new algorithm we propose are presented. Next, the implementation of an appropriate projection unit and an optimized scheduling for it are described. Finally, the validity of the algorithm is proven with illustrative results.

2.4.1 Perspective Projection

In this section we present both the classical approach and the new modified algorithm we propose for point perspective projection. The modified perspective projection solution takes advantage of the proximity of the projection on the screen of adjacent points in object space, inherent to the randomized z-buffer algorithm. This way, the projection of a given point can be easily

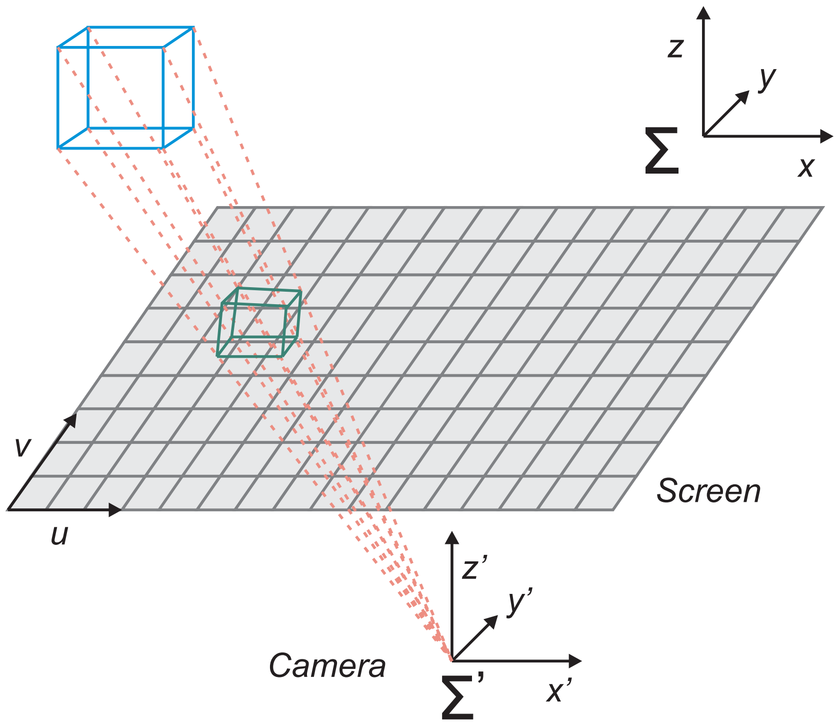


Figure 2.6: Perspective projection. Points are mapped from object space (Σ) to camera space (Σ').

computed employing the information relative to the projection of a neighbor point. The projection operation can be then converted into a simple selection operation among a low number of candidate pixels. Consequently, this technique allows high speed projections avoiding the utilization of the recursive dividers available in the graphics card.

2.4.1.1 Perspective Transformations

The rendering operation starts by computing, for each point, the coordinate transformation from object space to camera space. By camera space we mean the coordinate system in which the camera is at the origin and the screen is on the plane $z = 1$. This geometry transformation can be written as:

$$\begin{pmatrix} x' \\ y' \\ w \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{2.1}$$

where (x, y, z) are the object space coordinates of the point to be projected; (x', y', w) are interpreted as homogeneous 2D camera space coordinates of that point; and the matrix contains the rotation, scaling, and translation components of the transformation. To compute the screen coordinates (u, v) from the camera coordinates, a division operation is performed:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x' \\ y' \end{pmatrix} \cdot \frac{1}{w} \tag{2.2}$$

Then, the values (u, v) are given by:

$$u = \frac{a_{00} \cdot x + a_{01} \cdot y + a_{02} \cdot z + a_{03}}{a_{20} \cdot x + a_{21} \cdot y + a_{22} \cdot z + a_{23}} \tag{2.3}$$

$$v = \frac{a_{10} \cdot x + a_{11} \cdot y + a_{12} \cdot z + a_{13}}{a_{20} \cdot x + a_{21} \cdot y + a_{22} \cdot z + a_{23}} \tag{2.4}$$

This transformation implies the computation of three parallel inner product operations and two divisions. These equations can be implemented in hardware in graphics cards [57] by means of floating point multiply-accumulate units (fMACs) and floating point divider units (fDIV). Due to the timing requirements associated with the recursive structure of the dividers implemented on the graphics cards, a different technique should be employed in point rendering algorithms, where a large amount of points have to be projected at much higher rates. Following, an alternative projection technique, which avoids the division operations, is proposed and analyzed.

2.4.1.2 Modified Perspective Projection Algorithm

The standard perspective projection algorithm implies two divisions per point. This operation represents the limiting factor of the projection operation, due to the recursive structure of current floating point divider units (fDIV). Therefore a new technique for a more efficient point perspective projection has been developed [1]. This approach is based on the proximity of the projections on the screen of adjacent points in object space. As a result, the division operations associated with the classical perspective projection algorithm are eliminated.

The new technique we propose is based on the regular data distribution of the recent out-coming point rendering proposals in the bibliography [99, 138, 130, 105]. In these algorithms, a regular sampling is performed and an efficient hierarchical structure is used to store the scene. A hierarchy is

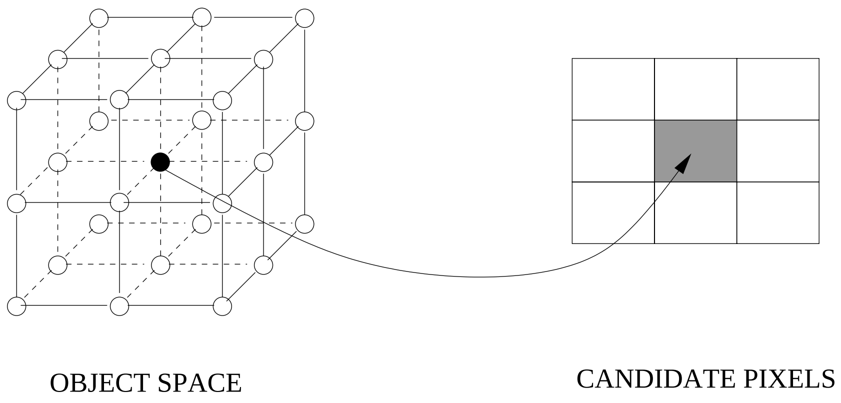


Figure 2.7: Projection of neighbor positions [1].

used where the scene is partitioned in different cubic resolution areas forming an octree structure. At each node there is a subsampled version of its children, the lowest resolution being stored at the root node. Specifically in the randomized z-buffer approach [130], a sufficient level of oversampling is mandatory so that every pixel receives the projection of, at least, one point of the scene.

In order to simplify the presentation, we consider a grid of 32 $\times$ 32 $\times$ 32 samples per node, while an extension to other grid sizes is straightforward. We refer to the set of points associated with each node as *cube*. During the rendering process (from a view point) the octree is recursively traversed from the root. The recursion process selects all nodes intersecting the view frustum and, to choose the octree level to be projected, it conservatively estimates for each node the number of points per pixel. The working set of the hierarchy is stored in the graphics board and reused many times over several frames.

In order to obtain a good reconstruction of the final image independently of the use of splatting techniques, the octree resolution to be represented is chosen in such a way that every pixel receives at least one point [130]. It is important to note that a consequence of this resolution level is that two contiguous positions in the grid are projected to contiguous pixels, if not the same pixel, on the screen. On the other hand, as the objects are modelled as a dense set of surface point samples, the points on a surface occupy adjacent positions on the grid and are then projected to equal or contiguous pixels on the screen.

Let us consider the example of Figure 2.7, where the projection of the central point (black point) and its first order neighbors are considered. Once the projection of the central point is determined (grey pixel), its first order neighbor locations are projected into the 9×9 pixel area indicated in Figure 2.7 (grey pixel and contiguous pixels).

Let us denote the screen coordinates of two contiguous points (x_i, y_i, z_i) and $(x_{i+1}, y_{i+1}, z_{i+1})$ as (u_i, v_i) and (u_{i+1}, v_{i+1}) respectively. In the following we will only consider the computations for the u screen coordinate, being the corresponding process for v completely equivalent. So if the projection of a point (x_i, y_i, z_i) is u_i:

$$(x_i, y_i, z_i) \xrightarrow{P} u_i \tag{2.5}$$

The u_{i+1} coordinate value of the projection of a contiguous point $(x_{i+1}, y_{i+1}, z_{i+1})$ can only be one of the following three values:

$$(x_{i+1}, y_{i+1}, z_{i+1}) \xrightarrow{P} \begin{cases} u_i - 1 \\ u_i \\ u_i + 1 \end{cases} \tag{2.6}$$

The identification of the correct projection value can be performed through the analysis of:

$$\begin{gathered} \frac{x'_{i+1}}{w_{i+1}} - [u_i - 0.5] \\ \\ \frac{x'_{i+1}}{w_{i+1}} - [u_i + 0.5] \end{gathered} \tag{2.7}$$

or, in order to avoid division operations, through the analysis of:

$$\begin{gathered} x'_{i+1} - u_i w_{i+1} + 0.5 w_{i+1} \\ \\ x'_{i+1} - u_i w_{i+1} - 0.5 w_{i+1} \end{gathered} \tag{2.8}$$

so that if both values are negative, the pixel $u_i - 1$ is identified; if the first one is positive and the second negative; the pixel is u_i; and finally, if both values are positive, the resulting pixel is $u_i + 1$.

This technique is attractive because it allows the projection of a point to be performed incrementally by using the projection of a first order neighbor point. This way, the projection can be computed through additions and comparisons, avoiding the costly division operations.

2.4.2 Implementation of the Perspective Point Projection Algorithm

In this section we present the hardware implementation of the algorithm we propose. This implementation can be efficiently mapped to a graphics card resulting in a fast projection algorithm as no divisions are performed. The simple structure of the projection unit and its low computational requirements permit the utilization of different projection units working in parallel to increase the projection rate. Furthermore, in order to increase the speed of the system, a pipelined structure is analyzed.

We focus our interest on the transformation from camera to screen coordinates (Equation (2.2)) where the division operation is critical. The transformation from object to camera space coordinates (Equation (2.1)) is not so relevant in terms of performance because it consists on multiplications and additions, operations for which current graphics cards have more hardware resources. Additionally, simplifications can be performed due to the regular structure of the data, making incremental strategies also possible [50, 51].

Two considerations have to be taken into account to optimize the utilization of the architecture. The first one is related to the recursive structure of the algorithm, so that the utilization of a pipelined strategy as a way of reducing the clock cycle has to be carefully analyzed. As will be shown later, the pipeline can be efficiently exploited through the utilization of adequate scheduling. On the other hand, the second consideration is related to the potential bottleneck associated with the bandwidth of the frame-buffer as it may restrict the scalability of the system. In this case, the utilization of local cached frame-buffers [66, 136] together with an efficient scheduling should be employed to assure the scalability of the processing rate.

The generic structure of the proposed architecture for the transformation from camera space to screen space (Equation (2.2)) is indicated in Figure 2.8(a). It consists of a candidate generator unit and an identifier unit. The inputs of the system are the camera space coordinates of the point to be projected $(x'_{i+1}, y'_{i+1}, w_{i+1})$. The candidate generator unit identifies the candidate pixels for the projection. This unit employs, as input, the projection (u_i, v_i) of the previous contiguous point (x'_i, y'_i, w_i) and offers as output all candidate projection pixels: $(u_i - 1, v_i - 1)$, $(u_i - 1, v_i)$, $(u_i - 1, v_i + 1)$, $\cdots$, $(u_i + 1, v_i + 1)$. The proper pixel (u_{i+1}, v_{i+1}) whereto the point is projected is selected in the identifier unit.

The detailed structure of the system is depicted in Figure 2.8(b). Specifically, only the computations related to the u coordinate are depicted, be-

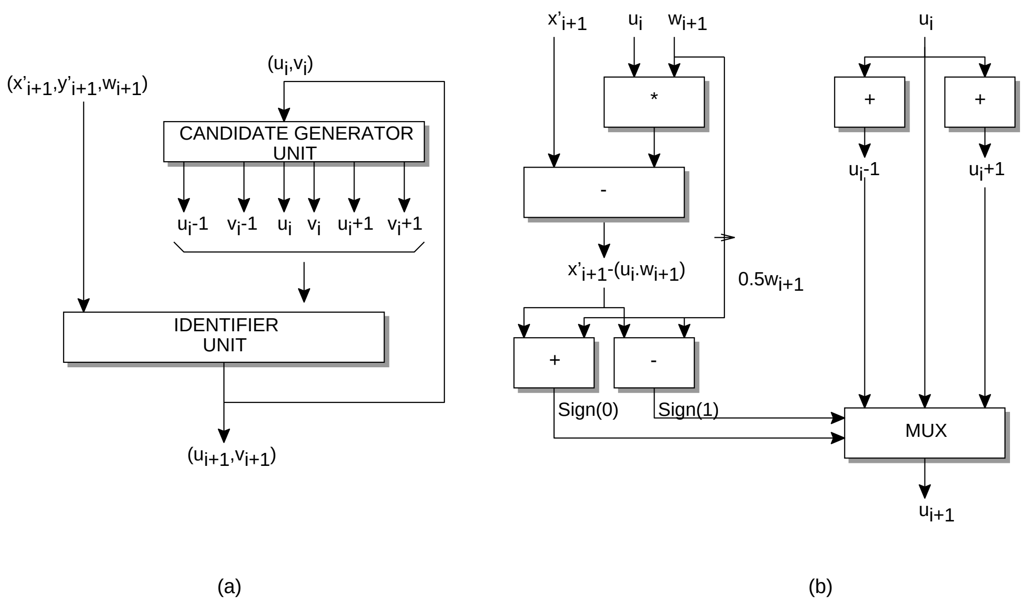


Figure 2.8: Projection architecture (a) Scheme (b) Detailed structure [1].

ing the computational structure associated with the v coordinate equivalent. The identifier unit is basically compounded of a multiplier, a subtractor, two simplified adders (where only the sign of the result is required) and a final multiplexer (mux), meanwhile the candidate generator unit consists of two incrementers. In the first multiplier and adder, the $x'_{i+1} - (u_i \cdot w_{i+1})$ value is computed. In the second level of adders the $0.5 \cdot w_{i+1}$ value is added and subtracted respectively. The resulting signs are employed in a mux to select the final screen coordinate value.

Let us emphasize the simplicity of the projection units that permits the utilization of several of them working in parallel to increase the projection rate. On the other hand, the avoidance of the division operations makes the resulting algorithm adequate to be implemented on current graphics cards.

Subpixel Precision Architecture

The above presented algorithm permits the identification of projection pixels. In some applications, in order to reduce aliasing artifacts, subpixel resolutions are considered [100]. Following, the modifications of the architecture necessary to include a 3×3 subpixel precision – the most commonly used – are indicated. Extensions to different precision levels can be directly extrapolated from the results presented here.

Adapting the previous algorithm to subpixel precision is straightforward as only a different number of candidate pixels, in this case subpixels, have to be considered. For the subpixel precision system, the set of equations to be evaluated, similar to those presented in Equation (2.8), are:

$$\begin{array}{ll} x'_{i+1} - u_i w_{i+1} + i \cdot w_{i+1} & \text{w}ith \;\; i = 0.5, 1.5, 2.5, 3.5 \\ x'_{i+1} - u_i w_{i+1} - i \cdot w_{i+1} & \text{w}ith \;\; i = 0.5, 1.5, 2.5, 3.5 \end{array}$$

to identify the appropriate subpixel among all possible candidates: $u_i - 4$, $u_i - 3$, $u_i - 2$, $u_i - 1$, u_i, $u_i + 1$, $u_i + 2$, $u_i + 3$, $u_i + 4$. Note that the u_i, v_i coordinates identify here subpixel coordinates and not pixel coordinates as previously.

This new set of equations can be implemented with a structure similar to the one presented in Figure 2.8(b) where, in this case, eight sign values have to be generated. It is important to note the simple scalability associated with the subpixel precision.

2.4.2.1 Scheduling Strategy

The simplicity of the projection operation and its low computational requirements permit the utilization of different projection units working in parallel to increase the projection rate. However, the parallelization of the algorithm can make the bandwidth of the frame-buffer to become a bottleneck, as different screen areas (frame-buffer areas) might have to be accessed simultaneously. This problem can be overcome by adopting a local cached frame-buffer solution [66, 136]. For this solution to be effective in terms of memory usage and processing performance the data should be distributed among projection units in such a way that the local frame-buffer size be minimized, i.e., that the projection of points assigned to the same projection unit falls into a small screen area.

On the other hand, in order to reduce the clock cycle of the system, a pipelined strategy can be utilized. However, the exploitation of the pipeline has to be carefully analyzed due to the recursive structure of the projection algorithm. This recursive structure introduces data dependencies which, if not handled carefully, might produce empty cycles (bubbles) in the pipeline, thus dropping performance. This suggests the simultaneous utilization of independent data produced from different seed points (the first point projected with the standard projection algorithm). However, such a choice of data for each pipelined unit would act as a limiting factor for the efficiency of the parallel solution. Therefore, a proper scheduling is mandatory in order to obtain the best benefit from both, parallelization and pipelining.

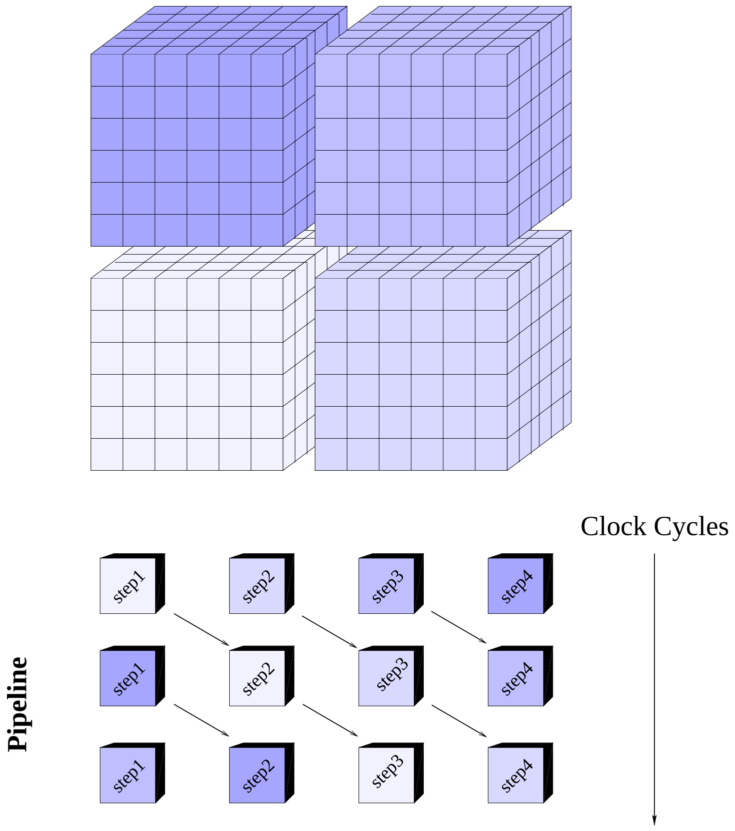


Figure 2.9: Cyclic distribution of cubes on the pipeline [1].

Following, a scheduling algorithm is presented that efficiently exploits a parallel system of pipelined units. The scheduling we propose permits, on the one hand, the exploitation of the parallel system, minimizing the size of the local cached frame-buffers associated with each unit and, on the other hand, an efficient utilization of the pipelined structure of each unit.

Firstly, let us consider the interleaving solution based on cyclic introduction of different cubes in the pipeline. To clarify, let us consider a four-stage pipelined projection unit where data corresponding to four different cubes are introduced in an interleaved manner. This is schematically indicated in Figure 2.9 where data corresponding to different cubes, depicted with different colors, are processed in a pipelined system with four stages. The projection of a point (x_i, y_i, z_i) from a given cube requires four cycles to be fully computed. The resulting projection value can be employed to project the neighbor point $(x_{i+1}, y_{i+1}, z_{i+1})$ of the same cube. This projection value is introduced into the first pipeline stage at the next cycle. To make possible the utilization of the proposed projection strategy, data corresponding to each cube has to be sorted in the CPU to exploit neighborhood information. That is, the points of each cube are sent from the CPU following a neighborhood sorting. Note that in order to start the recursive computations, the CPU has to perform the exact projection of one seed point for each surface (Equations (2.3)and (2.4)) to start the recursive computations.

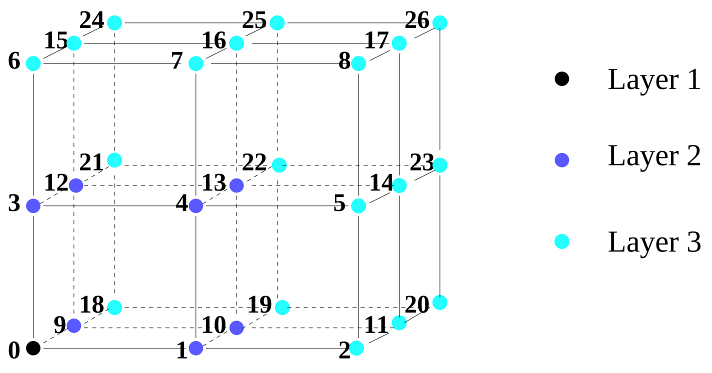
Therefore, introducing data corresponding to different cubes in an interleaved manner permits the filling and exploitation of the pipeline. However, the association of points from different cubes to the same unit can be a problem if the performance of a local cached frame-buffer is considered. Managing data from different cubes implies a larger projection area on the screen. Then, the larger the working screen area covered, the larger the local cached frame-buffer that should be employed, hence reducing its performance.

In order to minimize the cached frame-buffer size, only data corresponding to one cube should be processed by each unit before starting with a new cube. Moreover, the following cube should be adjacent to the former one, in such a way that the locality of the screen projections is exploited. This way, the coverage of only a small screen region is assured, so that the cached frame-buffer size is minimized.

Consequently, the scheduling we propose fills the pipeline stages of each projection unit with data corresponding to the same cube. The scheduling is based on the contiguity of the sampled points on a surface and on the identification of the different neighbors associated with each point. Specifically, let us interpret the result of the projection of a point as the seed for the projection of a neighbor point. The scheduling we propose is based on the computation of the different neighbors associated with the same seed and on the utilization of different seeds, corresponding to the same surface, simultaneously.

Next, and in order to simplify the presentation of the technique, we will consider that each cube has data corresponding to a unique surface. However, extending the results to different surfaces is straightforward. The scheduling requires a certain data sorting to be performed by the CPU so that the points are organized according to a neighborhood relationship. This sorting covers all points of the surface in an efficient way and without overlapping. The sorted data, organized in lists, are sent from the CPU to the graphics card where the projection computations are performed.

The basic idea, outlined in Figure 2.10(a), consists in distributing the grid cube structure in layers of connected points. In the Figure, for the sake of clarity, each point is labelled with a number and each layer is marked with a different color. The first layer comprises only one point, point $\{0\}$. The next



(a)

Layer 1 $\longrightarrow$ 0_7

Layer 2 $\longrightarrow$ 13_{19} 10_0 4_0 1_0 3_0 9_0 12_0

Layer 3 $\longrightarrow$ 2_0 11_0 5_0 14_0 18_0 19_0 20_0 21_0 22_0 23_0

6_0 7_0 8_0 15_0 16_0 17_0 24_0 25_0 26_0

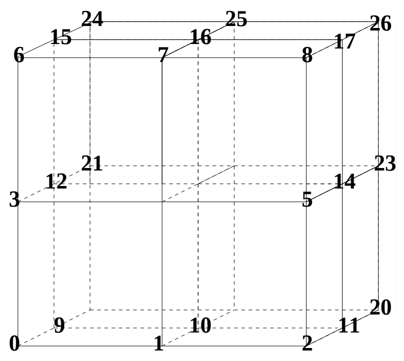
(b)

Figure 2.10: Data sorting. (a) Cube structure. (b) Layer representation [1].

layer is constructed with the neighbor points of that point, {13,10,4,1,3,9,12} and so on.

The data corresponding to each layer are organized in Figure 2.10(b) forming different lists. In order to optimize the scheduling, a data sorting has been performed inside each layer. Specifically, the data are organized according to their connectivity with the following layer, from higher to lower connectivity values. For example, point {13} in Figure 2.10(b) is listed before point {10} because it is connected to nineteen points of the following layer, while point {10} is only connected to ten points. On the other hand, in the following layer the data are organized according to their seeds, and to their own connectivity to the following layer. For instance, points associated with point {13} are listed first and organized according to their connectivity with the following layer. Note that in this example all ten neighbors of point {10} are already included in the list of neighbors of point {13} and therefore point {10} is labelled with 0 further neighbors. This way, the sorting of the points in a cube is based on the first order connectivity to the following layers.

Once the data are ordered in layers, the seed values to be employed



(a)

Layer 1 $\longrightarrow$ 0_5

Layer 2 $\longrightarrow$ 12_7 10_6 3_0 9_0 1_0

Layer 3 $\longrightarrow$ 16_3 6_0 15_0 7_0 21_0 24_0 25_0 14_0 2_0 11_0 5_0 20_0 23_0

Layer 4 $\longrightarrow$ 8_0 17_0 26_0

(b)

Figure 2.11: Sorting example. (a) Surface. (b) Layer representation [1].

for each point are identified. Each value indicated as a subindex in Figure 2.10(b) represents the new points of the following layer connected to the current point, but still not associated with a previous seed. As an example, point {13} can be employed as a seed for the computation of its nineteen neighbors in the following layer. On the other hand, point {10} is not employed as a seed for any point, since all its associated neighbors have already been computed as neighbors of point {13}.

In order to complete the presentation, let us now consider the real data distribution associated with sampled surfaces. In this case, although the continuity of the surface is assured, not all grid positions contain points. Here, instead of full layers, layer sections have to be considered. As an example, let us consider the point distribution represented in Figure 2.11(a) where the point positions are again labelled with numbers. This set of points can be classified in four layers as indicated in Figure 2.11(b). The first layer comprises, as usual, only one seed point, point {0}. The connectivity of the elements in the second layer {12, 10, 3, 9, 1} is given by the subindices {7, 6, 0, 0, 0}. This means, for instance, that the first point in the second layer, point {12}, is connected to the first seven points in the third layer {16, 6, 15,

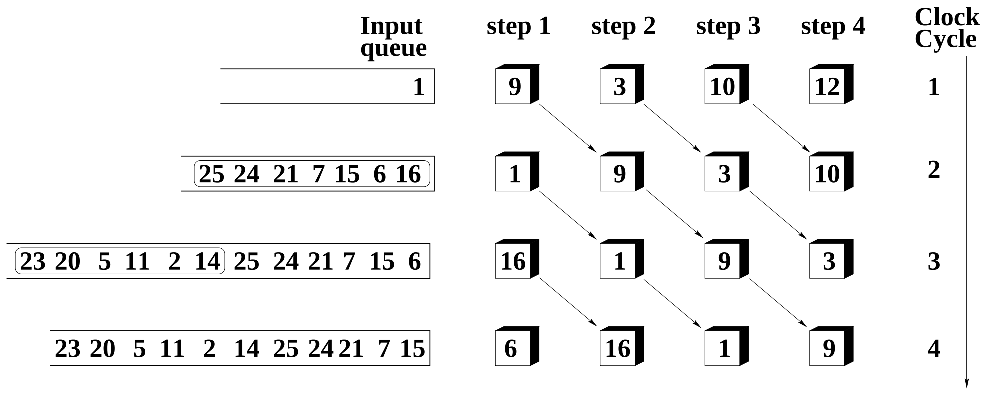


Figure 2.12: An example of the scheduling strategy [1].

7, 21, 24, 25}, meanwhile point {10} has six additional neighbors still not considered in the third layer list {14, 2, 11, 5, 20, 23}. These connectivity values permit the identification of seed values that can be employed for the projections. As an example, the point labelled as point {8} is the first one in the list of the fourth layer. Observing the information about the connectivity of the third layer to the fourth layer, it can be seen that the seed to be employed for projecting point {8} is point {16} of the third layer.

Once the points are sorted in the CPU, the resulting layer lists are sent to the graphics card where they are stored and processed. Taking into account that the connectivity information indicates the seed point that can be employed for the projection of each new point, the scheduling is directly determined by this representation of lists of layers. Assuming that the projection of the only point in the first layer list is already computed (by the CPU), the points of the second layer list can be introduced into the pipeline. When the first projection value associated with the second layer list is finished, the connected values of the third layer list can be introduced into the pipeline. We assume the utilization of an input queue for the temporal storage of data points whose seed has already been processed but cannot be introduced into the pipeline in the current cycle (e.g., in case the pipeline is already full). That is, if a point with connectivity n is processed, n new data from the following layer list can be added to the input queue for being processed.

In order to further clarify the functioning of the scheduling strategy, let us consider the example presented in Figure 2.12 where a four-stage pipeline unit is depicted. The points utilized for this example correspond to the data sorting described in Figure 2.11. Each row in Figure 2.12 represents the state of the input queue and the different stages of the pipeline along four consecutive clock cycles. Specifically, the evolution of point {9} is fully indicated from the first (cycle 1) to the last pipeline stage (cycle 4). In the first cycle, the data corresponding to the second layer list {12, 10, 3, 9, 1} are distributed along the pipeline and the input queue. Once the computation of the first value {12} is finished (cycle 2), since its connectivity value is 7, the first seven points {16, 6, 15, 7, 21, 24, 25} from the third layer list are appended to the input queue. In cycle 3, all points in the third layer list derived from the seed point {10} are also introduced into the queue {14, 2, 11, 5, 20, 23}. This process is repeated until all points in the current cube have been projected. Then a new cube is selected and the same strategy is started again until all points in the scene have been projected.

This way, this simple scheduling strategy makes use of an efficient sorting of the data performed by the CPU. Once the processed data are sent to the graphics card, the scheduling is based on the utilization of different seeds and on the computation of different neighbors associated with each seed. As a result, proper management of the data assures an efficient utilization of the pipeline structure with a simple and basic control, as will be seen in detail in the following section.

2.4.3 Results

Current graphics cards are optimized for processing triangular meshes where the projection operation is not critical. That is the reason why only few dividers, usually with a recursive structure, are available [57]. This structure is not adequate for point rendering applications, specially for the randomized z-buffer algorithm, making the projection algorithm to become an unsolved problem. In this sense, our novel method, presented in the previous sections, permits an increase in performance for the projection of points. By avoiding division operations, this strategy represents an interesting and efficient alternative to the standard projection algorithm for point rendering systems. Moreover, given the characteristics of our method it allows the utilization of parallelism, with the consequent boost in the resulting processing rate. On the other hand, the scheduling we propose permits the efficient utilization of a pipelined structure which helps to better exploit the capabilities of the available hardware.

We have implemented our projection solution through a software simulation. Subsequent rendering of the scenes has been realized according to the randomized z-buffer algorithm. Figure 2.5 shows point rendering scenes used to demonstrate the validity of the proposed method: Flock (Figure 2.5(a)), Landscape (Figure 2.5(b)) and Bunny (Figure 2.5(c)). All images were ob-

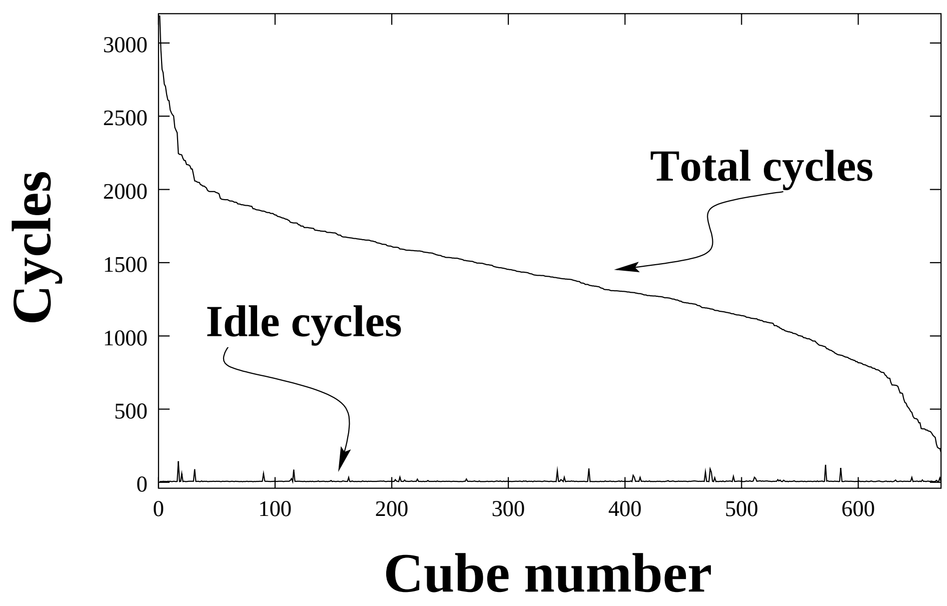


Figure 2.13: Total and idle cycles per cube [1].

tained with the projection and rounding of the points to the closest pixel position and written to a z-Buffer and color buffer. In order to improve the quality of the images a subpixel resolution may be considered.

Independently of the image quality, more related to the point rendering algorithm itself than to the projection technique employed, a key factor for analyzing the value of the proposed projection strategy is the efficiency achieved by the system using the above described scheduling. This is specially critical in this case, given the pipelined structure of the system. In Figure 2.13, the efficiency of the pipeline for the scene in Figure 2.5(c) is analyzed. The two curves in this figure represent the total number of cycles required per cube and the number of idle cycles respectively. In this simulation, the pipeline comprises five stages. Note that the number of idle cycles corresponding to each cube is very small resulting in a 99.32% efficiency ($\frac{total\ cycles - idle\ cycles}{total\ cycles}$). These results were obtained employing local interleaving (applied to different surfaces corresponding to the same cube). This way, the processing rate of the projection is mainly determined by the degree of parallelism and the pipeline levels employed.

2.5 Summary

In this chapter we have presented a novel approach to efficient hardware-accelerated projection applied to point rendering applications. The focus in our work has not been to improve the visual quality of scenes rendered with points as primitives, but to provide the steps necessary to achieve interactive or even real-time frame rates in the visualization of complex scenes modelled as sets of points. For rendering, our system is based on the randomized z-buffer algorithm [130] which relies on a high level of oversampling. However, it can also be applied to any other point rendering solution by applying just minor changes in the sorting of the data. The above presented method is an incremental strategy based on the (standard) projection of an initial seed point and the identification of pixel candidates for the projection of subsequent neighbor points out of the seed one. The corresponding next correctly projected point is then selected through simple comparison operations. This simplicity permits the implementation of high speed projections where no dividers are required, resulting in an adequate structure for being mapped into a graphics card.

To increase the speed of the projection operation and due to the low computational requirements associated with the projection operation, a parallel solution can be employed. An appropriate scheduling strategy permits to exploit the parallelism associated with point rendering where points can usually be treated individually. Furthermore, we have proven that the design of the scheduling successfully deals with the bottleneck created by the recursive structure of the proposed projection algorithm. This way, a parallel pipelined system can be efficiently realized. The architecture is also simple and regular, which assures good scalability. As a result, a high processing rate, necessary for highly complex scenes, can be achieved, hence contributing to improve the performance of hardware-accelerated point rendering applications.

While point rendering has steadily grown in importance as an alternative for rendering large, complex scenes, it is not, by far, the only valid possibility in modern computer graphics. The use of triangles as a modelling primitive remains as the standard solution for most applications. The study of efficient techniques to improve the quality and performance of the visualization of large triangle meshes is the topic addressed in the following chapter.

CHAPTER 3

Indirect Volume Rendering

3.1 Introduction

Despite the increasing importance being acquired by other rendering paradigms such as point rendering (see Chapter 2), or the use of implicit representations [29], volume rendering remains as the standard and most employed alternative for the visualization of volumetric data. Practically all volume rendering algorithms can be classified within two big groups: *indirect* and *direct* volume rendering. Direct volume rendering algorithms perform a view dependent interpolation and composition through the volume, normally combining different transparency and colors in order to give an insight of the rendered dataset. Since direct volume rendering and its applications will be discussed later in Chapter 4, we refer to there for a detailed description. On the other hand, in indirect volume rendering an explicit representation is generated for each element in the volume (e.g. bones, blood vessels, etc. in a medical dataset) that is to be visualized. Typically, this representation consists of a polygonal (triangle) mesh modelling the outermost surface of the object or feature of interest. Triangle meshes are normally used due to their suitability for being rendered efficiently on standard graphics cards. The surfaces represented by these triangle meshes are usually known as *isosurfaces*, as they approximate the shape of a surface delimitated by the position of all points in the volume with a given *isovalue*. Therefore, two are the main issues to be addressed in indirect volume rendering: first, the identification and generation of the isosurface/-s and second, rendering the obtained isosurface. In the remainder of this chapter, relevant algorithms dedicated to solve these two problems are presented.

While this project has been fully created and implemented by the author of this thesis, the collaboration of Jan Fischer on early design stages and many fruitful discussions should also be acknowledged.

3.2 Isosurface Extraction

Most scanning devices used in medicine, such as computed tomographs (CT) or magnetic resonance scanners (MR), produce volume datasets sampling a continuous magnitude at discrete points on a regular grid. Let $\mathbf{r_{i,j,k}} \in \mathbb{R}^3$ with $i = 0, ..., i_{max}$, $j = 0, ..., j_{max}$ and $k = 0, ..., k_{max}$, be the position of the grid points composing the volume (voxels). Since medical scanners generate a representation of scalar physical magnitudes (e.g., Hydrogen density), a scalar value $I_{i,j,k} \in \mathbb{R}$ is associated to each grid position. In this representation, a cell $C_{i,j,k}$ in the 3D regular grid has eight voxels at its corners forming a cube – isotropic datasets – or a prism – anisotropic datasets (the anisotropy is mostly only along the scan direction in medical datasets). An isosurface corresponding to an isovalue $c \in \mathbb{R}$, can then be defined as the solution set of the equation

$$\phi(\mathbf{r}) - c = 0 \tag{3.1}$$

where $\phi(\mathbf{r})$ is a continuous interpolation function $\phi : \mathbb{R}^3 \mapsto \mathbb{R}$, such that $\phi(\mathbf{r_{i,j,k}}) = I_{i,j,k}$ for all voxels $\mathbf{r_{i,j,k}}$.

Therefore the problem of finding an isosurface can be decomposed into two main tasks: a first step where the cells containing an intersection with the surface are identified, and a second one responsible for generating the corresponding triangulation from the selected cells.

3.2.1 Marching Cubes

Probably the best-known and still mostly used isosurface extraction method is the *Marching Cubes* algorithm introduced by Lorensen and Cline [82]. The *Marching Cubes* algorithm uses a divide-and-conquer approach which simplifies the global problem of extracting an isosurface from a complete 3D dataset, to obtaining extracts of that isosurface in a single cell basis. While many optimizations to the original algorithm have been proposed along the years, this divide-and-conquer strategy remains the core of almost every single isosurface extraction method.

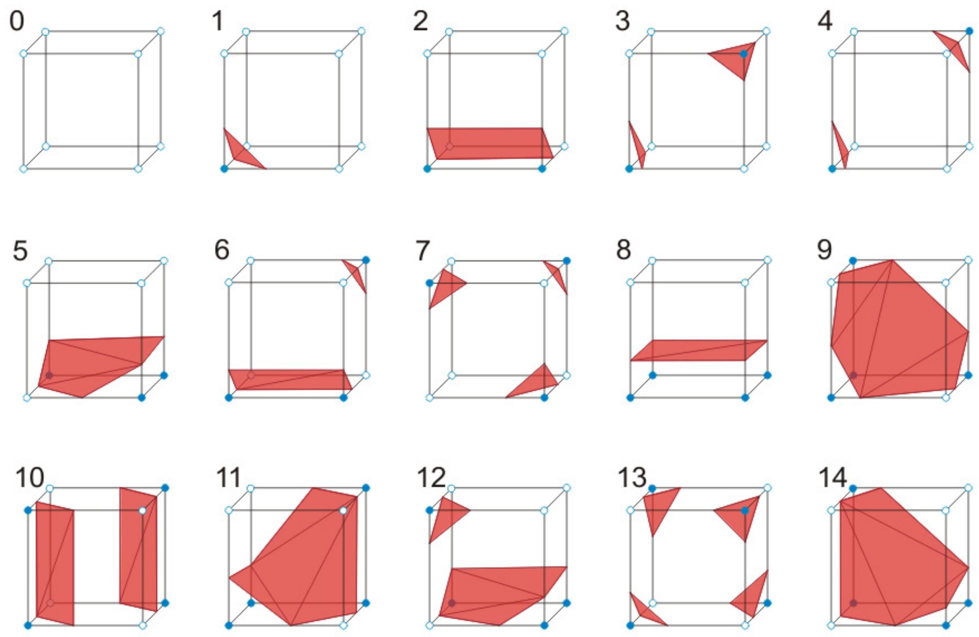


Figure 3.1: Marching cubes: 15 simplified triangulation cases.

The algorithm determines how the surface intersects each volume cell $C_{i,j,k}$. The surface intersection is identified by comparing the intensity of each of the eight voxels forming the vertices of the cell with the sought after isovalue. If the intensity exceeds (or equals) the isovalue, the voxel is inside (or on) the surface. Otherwise, if the intensity is lower than the isovalue the voxel is outside the isosurface. By assigning 1 to each interior voxel and 0 to each exterior one, all possible combinations for a cell can be precomputed. Since there are eight vertices in each cell and two possible states for each of them, there are only $2^8 = 256$ possible ways for a surface to intersect one volume cell. However, by taking into account the interchangeability of complementary cases, where interior and exterior vertices are respectively switched without modifying the topology of the triangulated surface, only cases with zero to four intersections need to be considered. Thus, the number of cases to be analyzed can be reduced to 128. Furthermore, if rotational symmetry is also employed, the total number of cases can be reduced until reaching the 15 cases shown in Figure 3.1. These 15 cases can be easily stored in a look-up table (LUT). This way, the result of the eight voxel intensity comparisons can be utilized to address the triangulation look-up table and find out the corresponding topology for the current cell. The exact position of each triangle vertex is then computed interpolating the surface intersection along the active edge. Typically, linear interpolation is used for this purpose, although higher degree interpolation schemes might be utilized too.

Finally, a normal vector at each triangle vertex is calculated for shading purposes. Normal vectors are approximated from the gradient values at the voxel positions forming the vertices of a cell. In this case, similarly to the procedure followed for the triangle vertex position, the normal vector is obtained (linearly) interpolating the gradient vectors at both extremes of the corresponding edge. Since in most volumetric datasets there is no gradient information directly stored with the data, the gradient, $\vec{G}(x,y,z) = \vec{\nabla} I(x,y,z)$ is normally estimated using central differences along the three coordinate axes:

$$\begin{aligned} G_x(i,j,k) &= \frac{I(i+1,j,k) - I(i-1,j,k)}{\Delta x} \\ G_y(i,j,k) &= \frac{I(i,j+1,k) - I(i,j-1,k)}{\Delta y} \\ G_z(i,j,k) &= \frac{I(i,j,k+1) - I(i,j,k-1)}{\Delta z} \end{aligned} \tag{3.2}$$

where $I(i,j,k)$ is the intensity at pixel (i,j,k) and Δx, Δy, Δz are the spacings along the three main coordinate axes.

Later research discovered an ambiguity problem in the original *Marching Cubes* look-up table [86, 94, 127] that could lead to a topologically inconsistent isosurface and the introduction of artificial holes. This inconsistency can be resolved by using a full look-up table containing the 256 possible cases, instead of the reduced 15 cases one.

3.2.2 Accelerated Isosurface Extraction

The *Marching Cubes* algorithm presented above clearly demonstrates that isosurface extraction can be satisfactorily solved using a divide-and-conquer approach. Furthermore, this method provides good quality results in terms of accuracy for most applications provided the resolution of the original volume data is high enough. However, with the growth of spatial and temporal resolution experienced by medical scanners (e.g., CT, MRI) and computer simulations in the last decades, also the size of the produced datasets has increased dramatically. The *Marching Cubes* algorithm is not well suited for fast isosurface extraction when dealing with large volumes, since all cells in the volume are visited in order to determine whether they are intersected by the isosurface or not. As only those cells which actually are intersected by the isosurface need to be visited and triangulated, the performance of the whole isosurface extraction can clearly benefit from an optimized search. Many different techniques have been proposed to accelerate this search phase. These can be classified within three main categories depending on whether they operate in geometric, image or value space decomposition [79]. Formally, this is indicated in Definition 3.2.1.

Definition 3.2.1 Geometric and Value Space

Let $\phi : S \mapsto V$ be a given scalar field and let D be a sample subset over ϕ, such that,

$$D = \{d_i\} \qquad d_i \in D = S \times V$$

where $S \subseteq \mathbb{R}^n$, with $n \in \mathbb{Z}$ (usually $n = 3$), is a *geometric space* and $V \subseteq \mathbb{R}$ is the associated *value space.*

Under these conditions, the problem of finding an isosurface through geometric search [79] can be defined as shown by Approach 3.2.1:

Approach 3.2.1 Geometric Search

Given a point $v \in V$ and given a set C of cells in S space where each cell is associated with a set of values $\{u_j\} \in V$ space, find the subset of C which an isosurface, of value v, intersects.

Following, the most relevant geometric space decomposition techniques, sometimes also called *space-based* methods, are presented.

Geometric Space Decomposition

Whenever a volume dataset relies on a structured grid as its internal structure, as it is still the case for most scanned medical data, spatial coherence can be exploited to optimize the localization and extraction of isosurfaces. One of the first attempts to address this issue was presented by Wilhelms and Van Gelder [133], who employed the branch-on-need octree (BONO), creating a hierarchical decomposition of the cell set. In their octree representation, each node stores information such as the minimum and maximum values of the cells it contains. This information is then utilized during the search phase, which consists of a traversal of the octree. This way, sections of the tree can be trimmed off during the search, thus reducing the number of visited cells. Livnat et al. [81, 78] analyzed the time complexity of this search phase, concluding a worst-case complexity of $O(k+k\ log(n/k))$, where n is the total number of cells and k is the size of the extracted isosurface. Nevertheless, the use of octrees is mostly devoted to structured grids, being an adaptation to unstructured grids not trivial.

Bajaj et al. [24] use set theory to find seed cells and a segment tree to organize and traverse them. This technique, which theoretically provides near-optimal worst case time complexity, is applicable for both structured and unstructured meshes. Its main drawback is, however, its sensitivity to noise in the volume dataset that may disturb the rather complex seed set construction process. Such noise is common in scanned data, e.g., CT or MRI scans, which causes the algorithm to produce a large number of seed cells, thus causing slower preprocessing times.

The *extrema graphs* proposed by Itoh et al. [59] use a geometric space decomposition to accelerate the search phase of the isosurface extraction process. This approach relies on the spatial coherence inherent to all isosurfaces, meaning that cells intersected by the isosurface are typically connected to each other. The search starts at a *seed* cell which is known to be intersected by the isosurface, and propagates recursively to its adjacent cells. An analysis of the way the current cell is intersected by the isosurface allows to guide the propagation so that only those neighbor cells which are guaranteed to be intersected too are subsequently selected.

An extrema graph is employed to find an appropriate seed cell from which the search can be started. The nodes of such a graph are cells including local extremum (minimum or maximum) values, while each arc of the graph has a list of the cells connecting its two end nodes. The use of these local extrema is based on fixed rules governing the relationship between an isosurface and a volume [61] (see Definition 3.2.2 and Remark 3.2.1).

Definition 3.2.2 Extremum Point

An extremum point is a node whose scalar value is higher (or lower) than those of all adjacent nodes that are connected to the node by cell-edges.

Therefore, given an isovalue, the extrema graph is traversed in order to

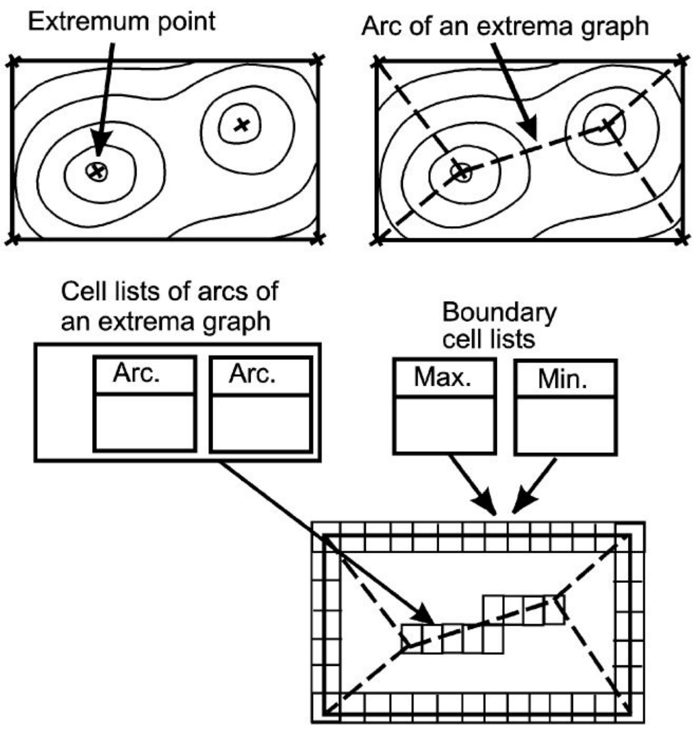


Figure 3.2: Extrema graph and boundary cell lists [61].

Remark 3.2.1

If there is a closed isosurface, then extremum points exist both inside and outside it. If there is an open isosurface, then it intersects the boundary of the volume.

locate arcs that span across that isovalue. The cells in each of these arc lists are then scanned sequentially until a seed cell is found. Boundary cells are registered as well and sorted in a list according to the minimum and maximum values of their voxels. These cells must also be traversed, hence the complexity of the algorithm is at best the size of the boundary list, which is estimated as $O(n^{2/3})$ by the authors. This makes this technique quite sensitive to the presence of noise in the dataset, producing small perturbations that might cause most nodes to be local extrema, raising the complexity of the algorithm to $O(n)$ in the worst case. This is, nevertheless, a rather unusual situation, only relevant with extremely noisy datasets.

Itoh et al. [60, 61] presented a further improvement to the original extrema graph technique introducing the *extrema skeleton* to avoid the dependency to the number of boundary cells. The initial extrema graph is replaced with a volumetric extrema skeleton obtained utilizing a thinning algorithm, an extension of a method commonly used in image processing. The volumetric skeleton preserves the topological features of the volume and the connectivity of the extremum points. It acts as a search index that facilitates the location of a proper seed cell. Once this has been found, the isosurface cells are propagated and consequently triangulated.

It must also be remarked that, even though the extrema graph is built based on intensity values, the search phase is guided by the spatial relationship between a selected cell and its neighbors, and therefore it is inherently a geometric space decomposition rather than a value space decomposition.

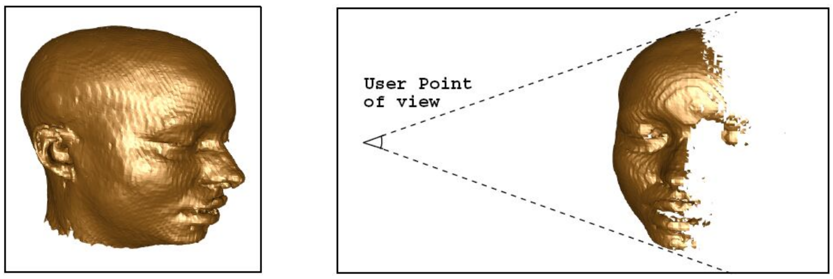
Image Space Decomposition

Due to the increasing size of today's datasets, such as high resolution medical scanned volumes or the results of flow simulations, which can easily reach several gigabytes, new problems are posed when it comes to the extraction and visualization of isosurfaces. The size of such an isosurface can arise to several million polygons, many of which are often projected to screen areas of under one pixel in size. In these cases, not only the computation of all the local triangulations can be very costly, but the huge number of polygons to be displayed can prevent a real-time or even interactive visualization. In such a scenario, several alternatives have been explored to optimize the visualization experience.

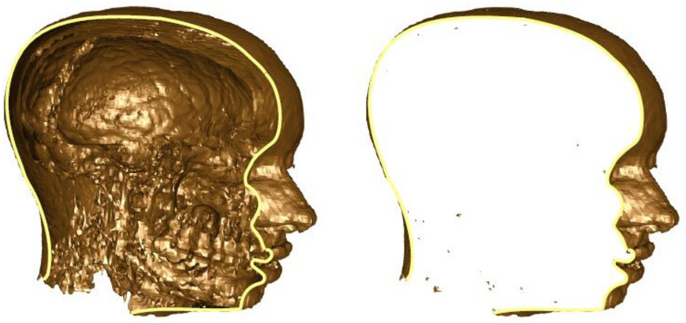
One approach is to use mesh reduction techniques [115, 95] to facilitate the rendering process. The mesh reduction can be applied during the isosurface extraction itself or afterwards as a postprocessing step [101]. However, mesh reduction is usually a computational intensive operation and requires extracting the whole isosurface previously for examination. Moreover, any change in the isovalue implies repeating the whole isosurface extraction and mesh reduction process over and over again.

Another possibility, which might be combined with mesh reduction, is the application of compression techniques to encode the polygonal mesh. A proper encoding of the mesh vertices and topology can lead to a noticeable rendering performance boost, specially when combined with hardware accelerated implementations [6] as will be presented in detail later in Section 3.3.

Finally, view-dependent isosurface extraction [80] focusses on reducing the time required for the search, triangulation and rendering of the isosurface all at once. This approach, as indicated by its name, tries to access only those



(a) Left: User view. Right: 90^o rotated view



(b) Cut plane through: Left: Full extracted isosurface. Right: View-dependent isosurface

Figure 3.3: View-dependent isosurface extraction [80].

cells producing triangles belonging to the visible portion of the isosurface, i.e., based on the image space. A hierarchical front-to-back traversal of the dataset needs to be performed for each view point, so that non-visible sections of the dataset can be skipped during the isosurface extraction. The potential benefits of this type of strategies are illustrated in Figure 3.3. Figure 3.3(a) shows a rendered scene from both the user view point and a 90^o rotated view position so that the actually rendered geometry becomes visible. This type of techniques are particularly well suited for those situations where only the outermost isosurface is of interest, thus saving the computational costs associated to render all complex inner structures, as can be seen in Figure 3.3(b).

Image space decomposition presents also two big disadvantages: visibility tests must be performed every time the view position is changed, which can easily be once per frame in interactive applications; and the savings are strongly reduced whenever transparency is introduced in the scene, since most structures become then visible.

Value Space Decomposition

A more general, and still effective, optimization of the isosurface extraction process is the decomposition of the value space, giving place to the sometimes also called *range-based* methods. By working directly with voxel values rather than spatial relationships, the underlying topology of the geometric structure is of no importance, making value space decomposition techniques applicable to both structured and unstructured grids. Furthermore, for scalar field datasets (in 3D), predominant in medical volume visualization, the dimensionality of the search field is reduced from three (X, Y, Z) to two (min, max). Even though the worst-case complexity of value space decomposition techniques is of $O(n)$, algorithms have been developed which considerable reduce this worst-case complexity.

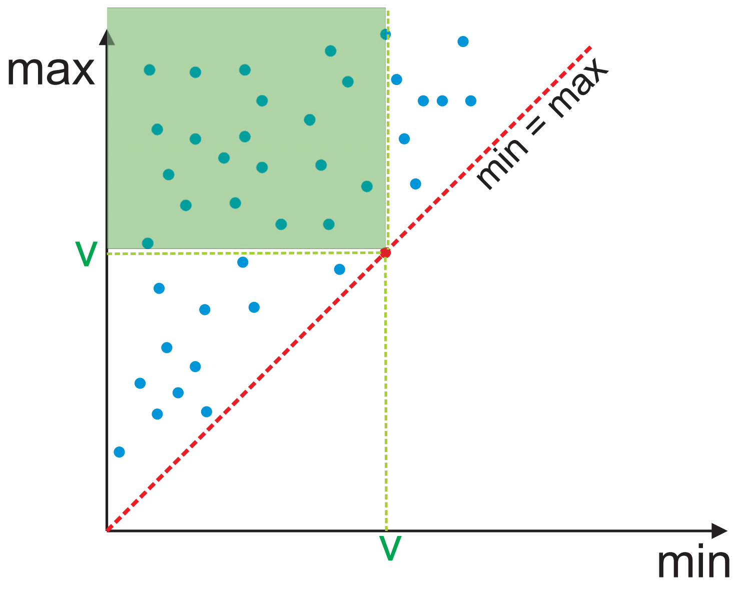


Figure 3.4: Span space representation.

Many value space decomposition solutions are represented by techniques based on the *span space* metaphor. Span space decomposition methods create and manipulate abstract representations of the cells based on their extreme values. The span space, first introduced by Livnat et al. [81], is a representation based on mapping each cell extreme values to a point in 2D space (V^2). In this 2D space the minimum scalar value of the cell is assigned to the x-coordinate and the maximum value determines the y-coordinate. The span space results very useful to geometrically interpret value space or range-based methods. Each cell can then be interpreted as an interval of intensities

$I = [a, b]$, which are represented as a point of coordinates (a, b). All points lie then automatically above the $y = x$ ($min = max$) line. The cells which contribute to an isosurface with isovalue, v, can be easily identified on the span space as the set of points lying to the left of the line $min = v$ and above the line $max = v$, as illustrated by Figure 3.4 with the green shaded region.

The span space, when employed as the basis for the search of active cells intersected by a given isosurface, provides the additional advantage of reducing the problem of searching over intervals of scalar values (voxel intensities in each cell) to a search over points in the (2D) span space. In order to further clarify these aspects, let us now formally define this augmented search space and its associated isosurface extraction search phase (see Definition 3.2.3 and Approach 3.2.2).

Definition 3.2.3 Span Space

Let C be a given set of cells, define a set of points $P = \{p_i\}$ over V^2 such that,

$$\forall c_i \in C \qquad associate, \qquad p_i = (a_i, b_i)$$

where,

$$a_i = \min_j \{v_j\}_i \quad and \quad b_i = \max_j \{v_j\}_i$$

and $\{v_j\}_i$ are the values of the vertices of cell i.

Approach 3.2.2 Span Search

Given a set of cells, C, and its associated set of points, P, in the span space, and given a value, $v \in V$, find the subset $P_S \subseteq P$, such that

$$\forall (x_i, y_i) \in P_S \quad x_i < v \quad y_i > v$$

Searching the intersected cells in the span space instead of performing other kinds of value space decomposition, such as an interval search (see below) has several advantages. One key concept is that points in 2D can be easily sorted according to their x and y coordinates. Compared to the case of intervals, specially overlapping intervals, this clearly facilitates the organization of the data in a hierarchical structure, thus easing the search phase too. Such a hierarchical data structure is employed by Livnat et al. in their *near-optimal isosurface extraction (NOISE)* algorithm [81]. The NOISE algorithm works with the span space as its underlying search domain and utilizes a *Kd-tree* as a means for simultaneously ordering the cells according to their maximum and minimum values.

Kd-trees were designed by Bentley [26] as a hierarchical data structure for efficient associative searching. Essentially, a Kd-tree is a multidimensional version of a binary search tree. Each node in the tree contains one data value and has two sub-trees as children. The division of nodes in the tree is performed in such a way that all nodes are ordered. This means that all nodes in one sub-tree, for instance the left branch of a root node, hold values which are lower than that of the root node, while those nodes in the right sub-tree correspond to values higher than that of the root node.

Another key characteristic of Kd-trees, which differentiates them from binary trees, is their multidimensionality. The data in the Kd-tree are sorted at each level of the tree according to alternating dimensions of the data. So, for the case of isosurface extraction for medical visualization using the span space as search domain, the Kd-tree structure alternates between the minimum, $x - min$ and the maximum $y - max$ values as the sorting dimension for each tree level. The construction of the Kd-tree can be carried out recursively in optimal time $O(n \log n)$. The tree construction problem consists in finding the median of the data values along the first dimension (e.g., *min*) and store it at the root node. The data are then partitioned according to the median and recursively stored in the two sub-trees. This process is subsequently repeated at each level of the tree alternating between the *min* and *max* coordinates for sorting and creating the sub-trees. As the Kd-tree has one node per cell or span space point, the memory requirements are trivially $O(n)$.

However, more important than the time complexity of the Kd-tree building process or its memory requirements, is the time complexity of the query. As already outlined above, a query within this context consists in, given an isovalue, v, locate all of the points in Figure 3.5, which are above and to the left of the horizontal and vertical lines at v respectively. In order to find out which points on the span space or cells in the volume satisfy this condition, the Kd-tree is traversed recursively starting from its root node. The isovalue is compared to the value stored at the current node, alternating between the minimum and maximum value depending on the current tree level. In the example presented in Figure 3.5, odd tree levels store minimum values, while maximum coordinates are hold at even levels of the Kd-tree. In this context, the first comparison is performed at the root node in terms of the minimum coordinates. Since this node is situated to the left of the isovalue, $x_{root} < v$, both sub-trees must be analyzed. Obviously if the minimum coordinate of

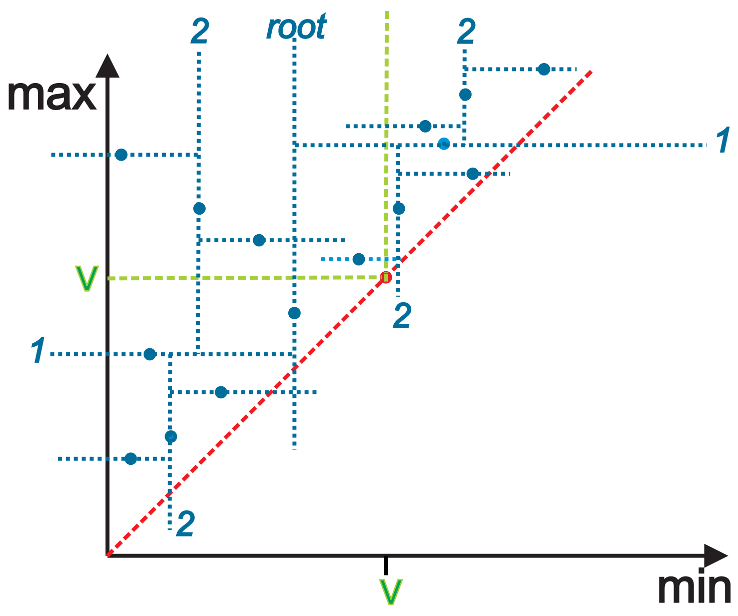


Figure 3.5: Query using a Kd-tree in a span space representation.

the root node had been greater than the sought after isovalue, only the left sub-tree would have been further traversed. At the next tree level, both sub-trees are hence inspected, being the maximum coordinates the ones to be compared to the isovalue. In this case, if the maximum (y) coordinate of the current node is less than the isovalue, $y_1 < v$, its lower sub-tree can be excluded from the search and only its upper sub-tree must be further traversed. Otherwise, both sub-trees should be traversed recursively.

This query system can be summarized in two search routines, *search-min-max* and *search-max-min*. The dimension being compared at the current level is named first, and the dimension to be analyzed at the next tree level is named second. The pseudo-code in Algorithm 3.2.1 indicates how the *search-min-max* routine works, being the *search-max-min* routine trivially symmetric to this one.

It is not trivial to estimate the time complexity associated to this query. Only some years after the work by Bentley introduced the Kd-trees [26], a worst case analysis was presented by Lee and Wong [73]. Obviously, for a given query, the time necessary is proportional to the number of nodes visited. The analysis of the worst case is based on a situation where all the visited nodes are not part of the sought after isosurface. In such a scenario, they showed that the worst case time complexity is $O(\sqrt{n}+k)$. The average case analysis is still an open problem. However, several studies [26, 112] sug-

Algorithm 3.2.1 *min-max* Search Routine

```
search-min-max( $isovalue$, $node$ )
if $node.min < isovalue$ then
   if $node.max > isovalue$ then
      triangulate $node$
   end if
   search-max-min( $isovalue$, $node.right$ )
end if
search-max-min( $isovalue$, $node.left$ )
```

gest that it is much faster than the worst case time $O(\sqrt{n}+k)$. Experimental results indicate that, for most applications, the number of active cells in a volume in relation to the total number of cells is $k \sim n^{2/3} > \sqrt{n}$ [79], which suggests a complexity of $O(k)$. The complexity of the isosurface extraction problem is $\Omega(k)$, as it is bound from below by the size of the output. Therefore, the NOISE algorithm is considered to be near optimal in the general case and optimal, for those cases where it is validated that $k \sim n^{2/3}$.

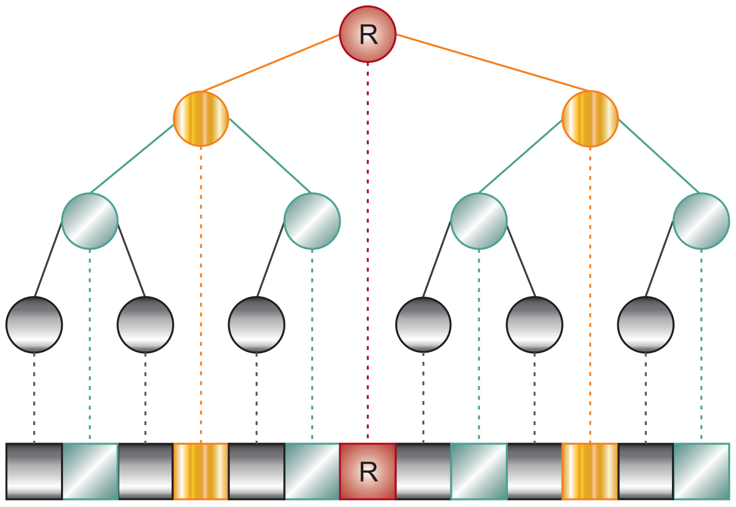


Figure 3.6: Pointerless Kd-tree.

Further improvements, such as the *pointerless Kd-tree* or an optimized search routine [81, 78, 79] can be employed to reduce the memory and search time corresponding to this algorithm. The former stores a Kd-tree as a one-dimensional array of nodes. In this array, the root node is placed at the middle, while the $(n-1)/2$ nodes representing the left sub-tree are at the first $(n-1)/2$ positions of the array, and the $(n-1)/2$ right sub-tree nodes are at the end of the array as shown in Figure 3.6. This produces a considerable reduction of the memory requirements, as only the minimum and maximum values of the cell must be held in each node together with one pointer to the actual cell data. The search routine can be optimized too, by distinguishing odd and even levels of the tree and taking into account previous comparison results. This is based on a trivial but useful remark derived from the examination of the previously presented *search-min-max* (*search-max-min*) routine (see Algorithm 3.2.1). As already indicated, if the current node's minimum is greater than the isovalue, the right sub-tree (containing nodes with minimum values greater than the current's one) can be directly trimmed. However, if the node's minimum is less than the isovalue, we still get information that is not being used by this routine. In this case, the minimum condition is automatically satisfied for all nodes in the left sub-tree. This means that, while we still need to recursively traverse the right sub-tree checking both dimensions, *min* and *max*, those nodes held in the left sub-tree only need to be compared to the isovalue in terms of their maximum coordinate. The optimized search procedure is illustrated by the routines in Algorithm 3.2.2.

The *search-max* routine in Algorithm 3.2.2 is only utilized with those nodes which already fulfill the minimum condition. Therefore, if the current node also fulfills the maximum value condition (i.e., $node.max > isovalue$), the current node's cell is triangulated, and the lower sub-tree (i.e., the sub-tree whose nodes hold maximum values less than the current node's maximum) is recursively traversed with the *search-skip-max* routine. On the other hand, we automatically can conclude that all the nodes in the upper sub-tree (i.e., the sub-tree whose nodes hold maxima greater than the current node's maximum) belong to the active area of the span space containing the cells intersected by the isosurface and consequently only have to be collected in order to be triangulated. This optimization in the search routine does not only reduce the traversal time, but also facilitates fast access to many of the intersected cells for their triangulation. This is possible due to the combination of the pointerless Kd-tree and the optimized search. Since in a pointerless Kd-tree any sub-tree is represented as a contiguous block of tree nodes, collecting all the nodes of a sub-tree requires only a fast sequential traversal of this contiguous memory sector, hence speeding up the access to the active cells.

An alternative way to approach the problem of isosurface extraction based on value space decomposition consists in orienting the search of active cells in the volume as the identification of intervals of the scalar magnitude (e.g., intensity) which contain the sought after isovalue. Under this approach, each

Algorithm 3.2.2 *min-max* Search Routine

```
search-min-max( $isovalue$, $node$ )
if $node.min < isovalue$ then
  if $node.max > isovalue$ then
    triangulate $node$
  end if
  search-max-min( $isovalue$, $node.right$ )
  search-max( $isovalue$, $node.left$ )
else
  search-max-min( $isovalue$, $node.left$
end if
search-max( $isovalue$, $node$ )
if $node.max > isovalue$ then
  triangulate $node$
  search-skip-max( $isovalue$, $node.down$ )
  collect( $node.up$ )
else
  search-skip-max( $isovalue$, $node.up$ )
end if
collect( $sub - tree$ )
sequentially triangulate all nodes in this $sub - tree$
```

cell is typically characterized as an interval of intensities $[a, b]$, where a and b are respectively the minimum and maximum voxel intensities in the cell. Taking this into account, an interval search can be formulated as indicated in Approach 3.2.3.

The *interval tree* introduced by Cignoni et al. [35, 34] works on this basis. This algorithm groups cells represented as the intervals, I_i, defined by the extreme values of the cell's voxels, $[a_i, b_i]$. These groups of cells constitute the nodes of a balanced binary tree, $\mathcal{T}$. For the formation of the tree, the extreme values of the intervals are sorted in a sequence of values, $\mathcal{X} = (x_1, ..., x_h)$ (i.e., each extreme a_i, b_i is equal to some x_j). Two lists are then held by each tree node, one sorted in ascending order of cell minima, $\mathcal{AL}$, and the other sorted in descending order of cell maxima, $\mathcal{DR}$. The root node has a discriminant value, $\delta_r = x_r = x_{\lceil \frac{h}{2} \rceil}$, so that the sorted sequence, $\mathcal{X}$ can be partitioned into three subsets as follows:

- $\mathcal{I}_l = \{I_i \in \mathcal{I} \mid b_i < \delta_r\}$. This subset represents the nodes composing the left sub-tree.

Approach 3.2.3 Interval Search

Given a point $v \in V$ and given a set of cells represented as intervals,

$$\mathcal{I} = \{I_1, \ldots, I_m\} \text{ such that, } I_i = [a_i, b_i], \text{ with } a_i \leq b_i, \text{ and } a_i, b_i \in V,$$

find the subset $\mathcal{I}_\mathcal{S}$ such that,

$$\mathcal{I}_\mathcal{S} \subseteq \mathcal{I} \quad \text{and} \quad a_i \leq v \leq b_i \quad \forall (a_i, b_i) \in \mathcal{I}_\mathcal{S},$$

where a norm should be used when the dimensionality of V is greater than one.

- $\mathcal{I}_r = \{I_i \in \mathcal{I} \mid a_i > \delta_r\}$. This subset represents the nodes composing the right sub-tree.
- $\mathcal{I}_{\delta_r} = \{I_i \in \mathcal{I} | a_i \leq \delta_r \leq b_i\}$. The intervals contained in this subset are held by the root node arranged into the two ordered lists previously mentioned: a list sorted in ascending order of cell minima, a_i, and a list sorted in descending order of cell maxima, b_i.

The left and right sub-trees are then recursively defined by taking the resting intervals and repeating the partitioning process for the extreme sets $(x_1, \ldots, x_{\lceil \frac{h}{2} \rceil - 1})$ and $(x_{\lceil \frac{h}{2} \rceil + 1}, \ldots, x_h)$ respectively. The interval tree can be constructed in $O(m \log m)$ time by a direct implementation of its recursive definition, producing a binary balanced tree with h nodes, and a height of $\lceil \log h \rceil$.

Figure 3.7 shows an example of the formation of an interval tree for a reduced number of intervals (11 cells). By construction, the tree has 11 nodes and a height of 4 levels. Due to the definition of the interval tree, the last level of the tree is generally empty, as it is the case here, since these nodes only might be generated by cells having the same minimum and maximum values (i.e., homogeneous cells).

For the extraction of an isosurface, given a query isovalue, v, the tree is traversed recursively starting at its root following Algorithm 3.2.3.

The interval tree and its corresponding search algorithm can be graphically interpreted using the span space representation. An example showing the span space equivalent for the intervals from Figure 3.7 is depicted in Figure 3.8. As can be seen in the figure, the discriminant, δ_r, defines a subdivision of the span space at each level of the interval tree. By the definition of the interval tree, intervals (i.e., cells) lying on a subdivision line belong to

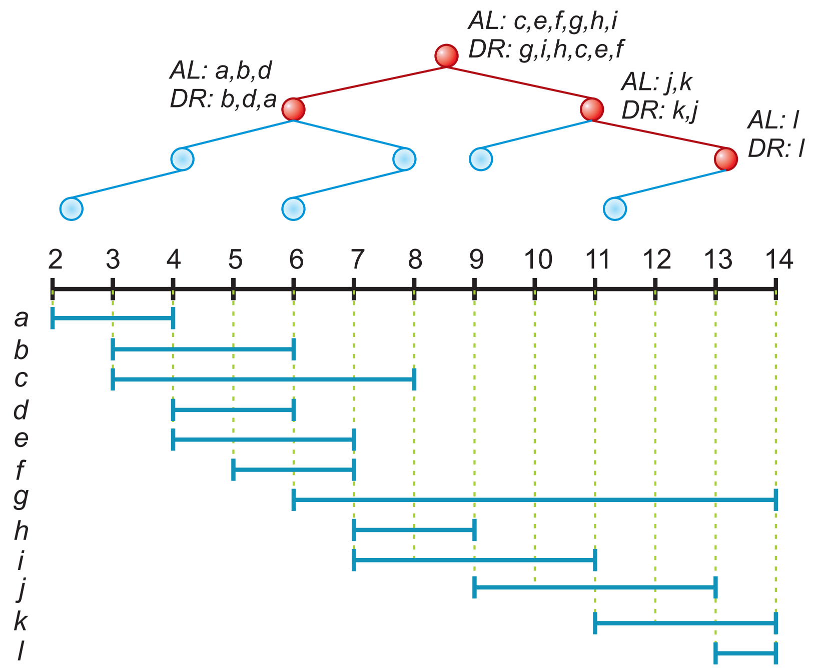


Figure 3.7: Example of an interval tree for a reduced number of intervals. The blue dots represent nodes with empty $\mathcal{AL}$ and $\mathcal{DR}$ lists.

the upper level of the tree. The tree search for a given isovalue, v, is also represented in Figure 3.8. Those space sections containing the horizontal dotted line that indicates the sought after isovalue, v, (i.e., sectors with $\delta_r \leq v$) are visited top-down (scanning the $\mathcal{AL}$ list) until the delimiting dotted line is reached. On the other hand, those sectors containing the vertical dotted line (i.e., $\delta_r \geq v$) are searched left to right (scanning the $\mathcal{DR}$ list) until the line is reached. Therefore, $\lceil \log h \rceil$ nodes of the interval tree are visited during a query, where h is the total number of distinct interval extreme values. Hence, for an output of size k, the computational complexity of the search is $O(k + \log h)$. It is important to note, that the time complexity of querying with the interval tree is not only optimal, but also independent of the total number of intervals (i.e., cells) and only dependent on the output size.

As this brief review of isosurface extraction acceleration techniques shows, there are many possibilities that have proven to reduce the latency associated to isosurface extraction with traditional *Marching Cubes*. However, despite this large extent of algorithms, not much information is available to assist the user in the selection of the method that best fits to a specific visualization problem. A case study performed by Sutton et al. [120] analyzes the benefits derived from several acceleration techniques in terms of latency and memory

Algorithm 3.2.3 Interval Tree Query

```
search-interval-tree( $isovalue$, $node$ )
if $isovalue < node.\delta_r$ then
  for all $\mathcal{I}$ in $node.\mathcal{AL}$ do
    if $\mathcal{I}.a \leq isovalue$ then
      select $\mathcal{I}$ as active
    else
      break
    end if
  end for
  search-interval-tree( $isovalue$, $node.left$ )
else if $isovalue > node.\delta_r$ then
  for all $\mathcal{I}$ in $node.\mathcal{DR}$ do
    if $\mathcal{I}.b \geq isovalue$ then
      select $\mathcal{I}$ as active
    else
      break
    end if
  end for
  search-interval-tree( $isovalue$, $node.right$ )
else
  for all $\mathcal{I}$ in $node.\mathcal{AL}$ do
    select $\mathcal{I}$ as active
  end for
end if
```

overhead. Even though no clear *winner* can be extracted from this study, due to the strong dependency existent between dataset and performance, it can be stated that the branch-on-need octree (BONO) approach [133] and the near-optimal isosurface extraction method (NOISE) [81] provide some of the best performances on average, being both comparable in terms of execution time. Of course, such a statement must be considered very carefully, given the high variance associated to the nature of the data and the isosurface sought after. Thus, factors like the level of noise, the spacial distribution of the surface within the volume, the size of the dataset, etc. can dramatically influence the performance obtained with each algorithm. Moreover, a fair comparison of all these different techniques is not easy and sometimes not even possible to perform, as many of them combine the core algorithm with additional optimizations. This way, value space decomposition methods can

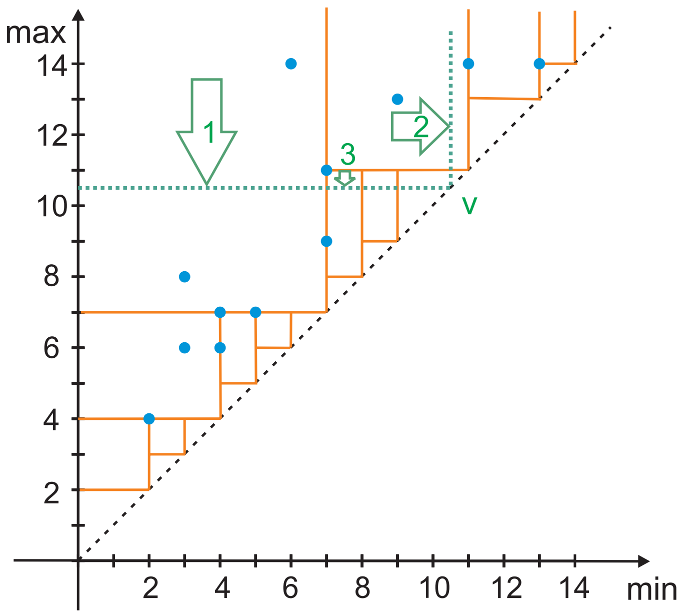


Figure 3.8: Graphical representation of the interval tree of Figure 3.7 in the span space. Solid lines indicate the partitioning of the span space according to the corresponding tree level. The arrows show the order followed during tree traversal for an isovalue v.

benefit from implementation optimizations related to the spatial coherence of the data (e.g. indexing of groups of cells) and, on the other hand, geometric space decomposition techniques make often use of value attributes as an extra hint during the search of active cells. Therefore, it is not always clear which technique is the most suitable for a general case, being necessary a personalized analysis of each specific problem to determine the most appropriate isosurface extraction algorithm.

The methods presented in this section represent the current state-of-the-art of acceleration techniques devoted to improve the extraction of isosurfaces from a volume dataset. In the remainder of this chapter we will now focus not on the isosurface extraction itself, but on how to render already extracted isosurfaces in a fast and efficient way.

3.3 Isosurface Rendering

The steady improvement in time and spatial resolution experimented by CT and MRI scanners during the last decades has lead to an important increase in the size and complexity of the datasets. Even though optimized search techniques can be utilized to accelerate the extraction of an isosurface from a large volume, the extensive number of triangles necessary to represent the obtained isosurface can produce a bottleneck in the rendering performance, thus reducing the interactivity of the visualization experience. This bottleneck is mostly due to the limited bandwidth of the bus connecting CPU and graphics card. Compression and encoding algorithms can be used to reduce the amount of memory necessary to represent the isosurface. The difficulty stems then from finding an efficient way to render the geometry out of its compressed representation. In this section, we present a brief survey of isosurface compression and encoding algorithms specifically devoted to overcome these limitations, as well as a detailed description of our own solution which was originally presented in [6].

3.3.1 Isosurface Compression

Standard algorithms for isosurface extraction, such as *Marching Cubes*, can produce an extensive amount of triangles when applied to large volume datasets employed for medical imaging. High resolution datasets usually generate large meshes with many small triangles. Even though surface simplification techniques [56, 28, 58] can reduce the complexity of these isosurfaces by removing and replacing vertices and triangles of the mesh, in some applications this alteration of the original data is not acceptable. This is the case, for instance, in medical applications, where the accuracy of the original data must be kept also in isosurface representations. An alternative approach to surface simplification is geometry compression. Given the copious amount of literature about compression techniques (see [40, 33, 52, 123] for examples of efficient mesh connectivity compression algorithms), here we restrict ourselves to those algorithms oriented to the compression of isosurfaces, which are of relevance for our work.

In recent years, Yang and Wu [137] described a method to compress triangle meshes generated by the *Marching Cubes* algorithm. Based on the fact that in isosurfaces generated by the *Marching Cubes* algorithm all vertices are placed along one edge of an active cell, it is granted that each vertex can be represented by a cell index, the index of the supporting edge, and its position along the supporting edge. The connectivity of the vertices, i.e., the topology of the mesh, is reconstructed by the decoder, in a rather complex process based on the 3D-chessboard structure first proposed by Cignoni et al. [34]. Since the *Marching Cubes* algorithm produces triangles whose vertices lie on an edge between two consecutive voxels, and each edge belongs

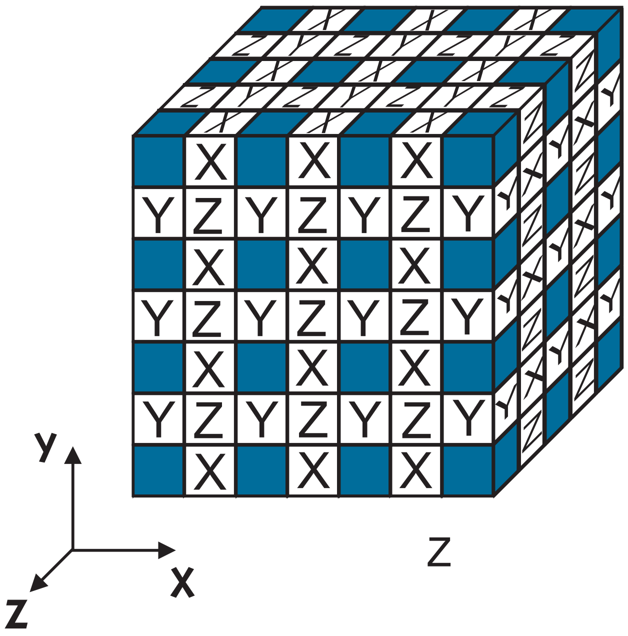


Figure 3.9: 3D-chessboard structure. All volume cells are classified into black and white cells. Black cells are explicitly encoded. White cells are labelled according to their closest black cell (X, Y, Z).

simultaneously to different cells (1–4 from boundary to interior edges), there is a considerable level of redundancy in the mesh representation it generates. The 3D-chessboard structure (see Figure 3.9) reduces this redundancy by considering only every second cell (*black cubes*) along each main direction (X, Y, Z) while taking into account neighborhood relationships to process the remaining cells (*white cubes*). This way, only a fourth of the total amount of cells is explicitly encoded, thus helping to reduce the memory requirements. Saupe and Kuska [110] presented an algorithm to compress isosurfaces that is also based on a similar vertex representation. In this case, the active cell set, often also called *occupancy image*, is encoded with an octree-based scheme in order to efficiently prune large homogeneous regions of empty space. Based on a similar isosurface representation, Taubin [122] developed a different approach that compresses the representation of the isosurface using a context based arithmetic coding scheme known as JBIG, typically dedicated to lossless compression of binary images.

In our approach [6], we employ an isosurface encoding representation analogous to that described by Saupe and Kuska [110] and also utilized by Taubing [122], which is presented in detail in Section 3.3.2. However, even though these approaches achieve considerable reductions in the memory size of the isosurface, they do not address the problem of rendering these encoded versions of the extracted geometry. Typically, the compressed isosurface must be decoded by the CPU and the set of polygons (generally triangles) are subsequently stored in main memory before being transferred to the graphics pipeline for being rendered, hence making bus bandwidth a major bottleneck. Such a hindrance can be overcome by moving the decoding stage from the CPU to the graphics card.

The main problem associated with the transfer of the decoding stage to the graphics card is that, at least until now, it was not possible to generate new geometry directly on the GPU. Every new vertex had to be explicitly defined and sent from the application (and therefore the CPU) to the graphics pipeline. Even though the creation of new geometry information directly on the GPU is still not directly supported, it is possible to reuse the functionality provided by current graphics cards supporting the DirectX Shader Model 3.0 (or its OpenGL equivalent), in order to overcome this limitation. Therefore, by applying the innovative strategy described in Section 3.3.3, it becomes possible to both decode and render an isosurface fully on the card. Specifically, in our method we make use of the newly available functionality on recent graphics cards (`GL_EXT_framebuffer_object`), which allows to *render* to off-line buffers and, with the assistance of fragment and vertex shaders, to reutilize the content of such buffers for subsequent rendering passes. This is the principle that allows us to *generate* geometry primitives with shader programs running on the GPU, without any read back to application memory. This constitutes, to our knowledge and at the time of first publication, the first such a solution for isosurface decoding and rendering, and constitutes the main contribution of this work.

As this has been a hot topic of research in the last years, other similar proposals have been also recently produced. A somehow alike solution has been presented by Krüger et al. [69] for rendering of large point scans with point rendering techniques. Also related, although not directly addressing the same problem, is the GPU-based implementation of isosurface extraction using the *Marching Tetrahedra* algorithm [41], which has been recently presented by Klein et al. [64]. Nevertheless, this solution, despite its efficiency in terms of performance, relies on the utilization of ATI's SuperBuffers functionality, from the proposed `GL_ATI_super_buffers` OpenGL extension [83]. This cumbersome extension has not been officially accepted by the OpenGL Architectural Review Board (ARB), which makes its functionality difficult to handle and restricts its availability.

3.3.2 Encoding Scheme

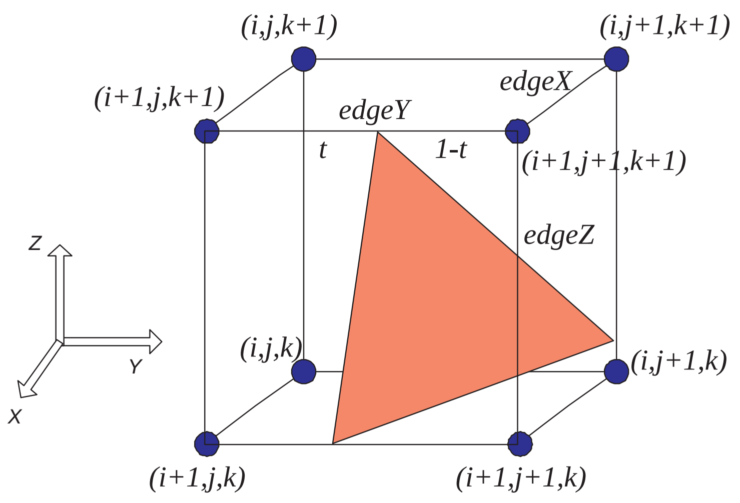


Figure 3.10: Volume cell representation. Each cell $C_{i,j,k}$ contains eight voxels [6].

Even though this encoding and decoding strategy can be efficiently utilized for rendering any triangle mesh in general, our work focuses on medical imaging applications. In this scenario, a 3D dataset is acquired by means of scanning devices (e.g., CT, MRI) as a discrete representation (sampling) of a continuous scalar magnitude (e.g., material density) at discrete points within the patient's anatomy. These samples are usually placed at the vertices of a regular grid (see Definition 3.3.1).

Definition 3.3.1 Regular Grid

Let $\mathbf{r_{i,j,k}} \in \mathbb{R}^3$ with $i = 0, \ldots, i_{max}$, $j = 0, \ldots, j_{max}$ and $k = 0, \ldots, k_{max}$, be the position of the points composing the volume (voxels). This set of points, $\mathbf{r_{i,j,k}}$, compose a regular grid if

$$\begin{aligned} r_{i,j,k} - r_{i-1,j,k} &= \Delta_x = \text{constant} \quad && \forall\ i = 0, \ldots, i_{max} \\ r_{i,j,k} - r_{i,j-1,k} &= \Delta_y = \text{constant} \quad && \forall\ j = 0, \ldots, j_{max} \\ r_{i,j,k} - r_{i,j,k-1} &= \Delta_z = \text{constant} \quad && \forall\ k = 0, \ldots, k_{max} \end{aligned}$$

(Note: A strict definition would also require $\Delta_x = \Delta_y = \Delta_z$, but this condition is often relaxed in medical imaging, as most medical data present $\Delta_x = \Delta_y \neq \Delta_z$).

As we work with scalar datasets, a scalar value $I_{i,j,k} \in \mathbb{R}$ is associated to each grid position. In this representation, a cell $C_{i,j,k}$ in the 3D regular grid has eight voxels at its corners forming a cube (see Figure 3.10). Let us now recall that an isosurface corresponding to an isovalue $c \in \mathbb{R}$, can be defined as the solution of the equation

$$\phi(\mathbf{r}) - c = 0 \tag{3.3}$$

where $\phi(\mathbf{r})$ is a continuous interpolation function $\phi : \mathbb{R}^3 \mapsto \mathbb{R}$, such that

$$\phi(\mathbf{r_{i,j,k}}) = I_{i,j,k} \tag{3.4}$$

for all voxels $\mathbf{r_{i,j,k}}$.

The first stage of the isosurface extraction process, presented in detail in Section 3.2.2, consists in finding those cells $C_{i,j,k}$, whose voxel intensities $I_{i,j,k}$ have values both above and below the sought after isovalue c. Such cells, usually denominated *active cells* or *intersecting cells*, are the only ones in the volume that contribute to the generation of an isosurface. Furthermore, in this scenario, all vertices of the generated mesh representing the isosurface are on edges of an active cell. In the standard case of *Marching Cubes*, vertex positions are linearly interpolated along one edge of an active cell according to the intensities of both voxels at the extremes of the edge, and the given isovalue. Taking this into account, we can build an alternative representation for isosurfaces, where instead of storing the 3D coordinates of each vertex (three floating point numbers, 32 bits each) and each normal vector (three floating point numbers, 32 bits each) of the mesh, indices identifying the vertex position within an active cell can be used. More specifically, a vertex position can be determined as

$$\mathbf{r} = \mathbf{r_{i,j,k}} + t_\alpha \mathbf{e}_\alpha \tag{3.5}$$

with $\alpha = x, y, z$, where e_α is a unit vector along one of the three main directions of the regular grid X, Y, Z, and $t_\alpha \in [0, 1]$ acts as linear interpolation factor from a voxel $\mathbf{r_{i,j,k}}$ to one of its first order neighbors along the edge selected by $\mathbf{e}_\alpha$. The value of t_α can be then expressed as

$$t_\alpha = \frac{c - \phi(\mathbf{r_{i,j,k}})}{\phi(\mathbf{r_{i,j,k}} + \mathbf{e}_\alpha) - \phi(\mathbf{r_{i,j,k}})} \tag{3.6}$$

At this point it is important to note that with this representation, each voxel position can identify up to six vertices along one of its six incident edges.

However, the half of these positions are redundant and can be assigned to one of its direct neighbors, by defining a scan direction. In this case, we associate to each voxel the three edges along the negative direction of the main axis, $\{e_{-x}, e_{-y}, e_{-z}\}$. This way, a vertex lying on one edge will be assigned to the *ceiling* voxel. This choice is arbitrary and for the sake of clarity, in the remainder of this presentation we will refer to the directions and edges as X, Y, Z instead of $-X, -Y, -Z$.

By using this representation, an encoded version of the isosurface can be created. Each vertex is now encoded as a 5-tuple of values $(i, j, k, t_\alpha, e_\alpha)$. Even though we have replaced the three coordinates of the vertex by five values, the memory requirements associated to the encoded version are considerably lower. The voxel indices (i, j, k) are integer values. In our implementation, these indices are stored using one byte per index, which allows to address volumes of size up to $256 \times 256 \times 256$. Larger volumes could be either split in blocks of this size, or encoded using 16 bits per index. The unit vector identifying the edge on which the vertex lies could be ideally encoded using only two bits, since only three values are possible. However, given that the encoded isosurface is to be transferred to the GPU as the content of a set of textures, the edge identifier must be stored using one byte, the smallest depth value of the supported texture format. Finally, the interpolation factor $t_\alpha \in [0, 1]$ is a real number that can be quantized for encoding. In our system, t_α is quantized with 8 bits and mapped to the range $[0, 255]$. This choice is sufficient for datasets acquired with a depth of 8 bits/voxel and does not introduce any extra uncertainty in the isosurface (see [110] for a complete discussion). Consequently, the position of each vertex is encoded using 5 bytes (40 bits) instead of the standard 12 bytes (96 bits), meaning that the encoded isosurface needs around 58% less memory than a standard OpenGL representation.

So far we have dealt with the encoding of the vertex positions. If necessary, it is possible to encode the normal vectors too. Since a normal vector is usually normalized to be a unit vector, its magnitude can be ignored, focussing only on encoding its orientation. This can be easily represented using the azimuth and zenith spherical coordinates (θ, ϕ), with $\theta \in [0, 2\pi)$, $\phi \in [0, \pi]$. Here again, (θ, ϕ) are real numbers that must be quantized in order to be efficiently transferred to the GPU in a texture. Different resolutions can be selected for the quantization of both components. In our implementation, we scale each value to be in the interval $[0, 255]$ and use 8 bits to encode each θ and ϕ value. Even though this introduces a certain error in the normal orientation, our tests show that this is negligible and does not result in visible artifacts. With such encryption, a normal vector can be encoded using 2 bytes (16 bits) instead of the 12 bytes (96 bits) of a standard representation, a reduction of more than 83%.

As briefly mentioned before, we have selected this representation, not only due to the reduction of memory requirements associated to the isosurface, but specially because of its suitability to be directly uploaded to graphics memory as the content of a set of textures. Specifically, two textures are employed to transfer the geometry information (vertex positions) and one texture is utilized to upload the encoded normal vectors, if required. The first geometry texture is a `GL_RGBA8` texture (8 bits/component), where the RGB components contain the (i, j, k) voxel indices, respectively, and the alpha component comprises the quantized and scaled interpolation factor t_α. The second geometry texture is even more simple and contains the edge identifier $\mathbf{e}_\alpha$ in a `GL_ALPHA8` (8 bits/component). Finally, the normal components, if needed, are saved as the content of a `GL_LUMINANCE8_ALPHA8` (8 bits/component) texture. This distribution of the data in three textures obeys not only to a logical separation of the encoded components, but also to the requirement that the chosen texture formats be directly supported by the graphics card to avoid undesired data transformations.

In order to obtain textures of the form $2^m \times 2^n$, with $m, n \in \mathbb{Z}$, the size of the textures is computed automatically depending on the number of vertices in the isosurface as

$$\begin{aligned} texture_{width} &= 2^{log_2 \lceil \sqrt{n_{verts}} \rceil} \\ texture_{height} &= 2^{log_2 \lceil (\frac{n_{verts}}{texture_{Width}}) \rceil} \end{aligned}$$

Alternatively, a rectangular texture (`GL_ARB_texture_rectangle`) can also be utilized, if supported by the employed graphics card and driver.

In either way, the textures containing the encoded isosurface can be generated and uploaded to graphics memory, from where the GPU-based decoder will access them to produce the vertices of the mesh ready to be rendered.

3.3.3 Decoding and Rendering

The fast development experimented by graphics cards in recent years, together with their increasing level of programmability has brought us to the point where GPU-based computations can clearly outperform CPU-based ones in scenarios where the highly parallel structure of the former can be fully utilized. We make use of this higher performance to accelerate the rendering of the encoded version of an isosurface presented in the previous section. Our solution is based on the newly available frame-buffer objects extension (`GL_EXT_framebuffer_object`), which allows to render to an off-screen memory buffer in the graphics card and to reuse this memory buffer as the source of a vertex array (an alternative to implement the so called *render_to_vertex_array* functionality). This way, a costly read back from graphics memory to main memory can be avoided, thus speeding up the rendering process.

The functional pipeline of our decoding and rendering strategy is the following:

- Upload data (geometry + normals) as textures.
- *Render* geometry (and normals) texture/-s to off-screen buffer/-s.
- Decode vertex position and normal vector using a fragment program and write results to a frame-buffer object.
- Bind content of frame-buffer object as source of a vertex array.
- Render vertex array.

Upload Data

The textures described in Section 3.3.2 are uploaded to graphics memory using OpenGL functions (`glTexImage2D`). This way, the geometry and the normals can be efficiently transferred to the graphics card as a whole. Note that with the given encoding scheme, only the information corresponding to the vertices must be sent to the graphics pipeline in order to render the isosurface, in contrast to GPU-accelerated isosurface extraction algorithms, where the whole volume must be copied.

Render Textures to Off-Screen Buffers

Once the textures have already been defined and their content has been uploaded, we need to compute the appropriate texture coordinates to access the correct texels for each vertex. This can be easily achieved by rendering a multi-textured quad of the same size as the textures. If a 2D orthogonal projection is defined and the viewport is also set to the size of the textures, every texel corresponds to a pixel on the screen, or as in our case, to a fragment in the off-screen buffer. This is important because, this way, a fragment program can be utilized to decode both the vertex position and the normal vector in a single pass. In order to do this, we must create a frame-buffer object with two different draw buffers (`GL_COLOR_ATTACHMENT0_EXT, GL_COLOR_ATTACHMENT1_EXT`). These act as two independent rendering targets similar to a regular frame-buffer, but without the precision restrictions associated to the latter. This is a crucial aspect, since floating point rendering targets can be defined, thus making it possible to *render* (write) the results of the decoding process (i.e., vertex coordinates) directly to a floating point buffer.

Decode Geometry

For decoding the isosurface on the GPU, a fragment program in the OpenGL Shading Language (GLSL) is used. Even though rendering the textures to off-screen buffers and decoding the geometry are two different logical steps, both are performed in one single pass. Therefore, the same shader program is run once to carry out both tasks. The shader program is formed by a vertex shader and a fragment shader. Given the adequate disposition of the projection matrix and the viewport, the vertex shader is extremely simple. It only must compute one texture coordinate (`gl_TexCoord[0] = gl_MultiTexCoord0`) and transform and orthogonally project the four corners of the multi-textured quad. The actual decoding process is executed by the fragment shader.

The fragment shader receives the volume parameters (volume offset and voxel's spacings) from the application, as well as the identifiers of the textures involved (two for geometry, three if normals are encoded). At the same time, the texel coordinate corresponding to the current fragment–vertex–texel is passed automatically from the vertex shader. With this information, the decoding process consists in first identifying the position of the two voxels on both extremes of the respective edge, and then interpolate between both positions using the linear factor t_α (see an illustration in Figure 3.11). The linear interpolation factor can be directly read from the texture as its value, alike all other fixed point precision texture values, is automatically scaled to be within $[0, 1]$. The rest of values read from the textures must therefore be scaled back to their original values with a multiplication by 255.

The position of the first voxel $\mathbf{r^1}$ can be computed as

$$r_i^1 = o_i + \Delta_i v_i \tag{3.7}$$

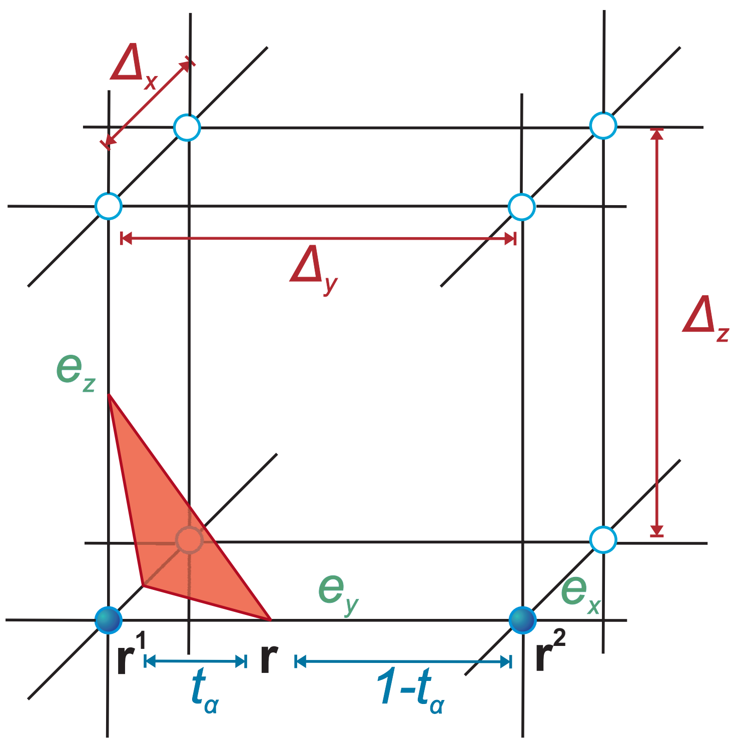


Figure 3.11: Illustration of the decoding process. The current vertex position is linearly interpolated between the position of the two voxels involved, $\mathbf{r^1}$ and $\mathbf{r^2}$.

where $i = x, y, z$, with o_i being the i-th component of the volume offset, Δ_i, the i-th component of the voxel spacing and v_i, the voxel index in the i-th direction read from the texture. Once the position of the first voxel has been computed and the corresponding edge $\mathbf{e}_\alpha$ subsequently identified by reading the respective texel value, the 3D coordinates of the second voxel $\mathbf{r^2}$ can be easily obtained as

$$\mathbf{r^2} = \mathbf{r^1} + \boldsymbol{\Delta} \cdot \mathbf{e}_\alpha \tag{3.8}$$

For the linear interpolation,

$$\mathbf{r} = t_\alpha \mathbf{r^1} + (1 - t_\alpha)\mathbf{r^2} \tag{3.9}$$

the OpenGL Shading Language function `vec3 mix(vec3 x,vec3 y,float a)` is used. This is a vectorial function that exploits the parallelism existent on the graphics board. The obtained vertex position is written as an RGB value in the first draw buffer bound to the frame-buffer object, where $R = r_x, G = r_y, B = r_z$.

On the other hand, the normal vector can also be decoded and the three cartesian components of the vector can be written to the second draw buffer bound to the frame-buffer object. The coordinates of the normal vector are computed from the encoded direction vector (θ, ϕ) as

$$\begin{aligned} n_x &= -cos(\theta)sin(\phi) \\ n_y &= -sin(\theta)sin(\phi) \\ n_z &= cos(\phi) \end{aligned} \tag{3.10}$$

Here again, these values are saved as the RGB content of a second draw buffer (off-screen buffer) with $R = n_x, G = n_y, B = n_z$.

This way, both the vertices of the mesh representing the isosurface and their respective normal vectors are written to graphics memory and ready to be used for rendering.

Render Isosurface

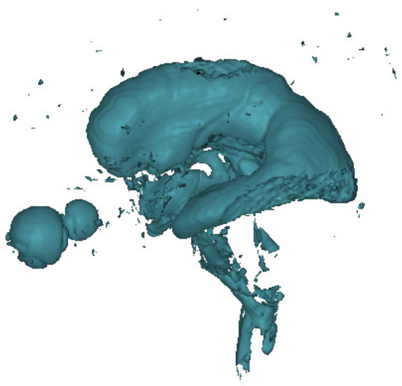
In order to render the isosurface, we reutilize the content of the two draw buffers where the geometry has been written to during the decoding process. This can be done making use of the `GL_EXT_pixel_buffer_object` OpenGL extension, together with the already mentioned `GL_EXT_framebuffer_object` extension. This allows us to bind each of the draw buffers of the frame-buffer object as the source for two vertex attributes of a vertex array. The first vertex attribute (identified by *index 0*) is mandatory and represents the vertex position, while the second is optional and can be used to define the normal vector of the corresponding vertex. Once these vertex attributes are bound by the application, the vertex array can be rendered using a second shader program so that each normal vector can be assigned to the built-in attribute `gl_Normal` in a vertex shader. A complete transcription of the vertex and fragment programs is presented in Appendix B.

3.3.4 Results

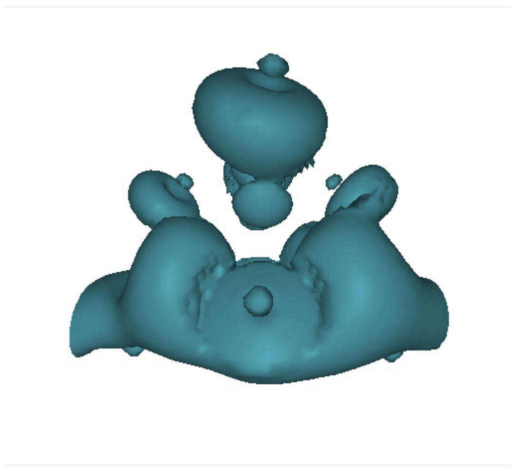
We have evaluated a prototypical implementation of the proposed method with a variety of datasets. Figure 3.12 shows three of these datasets, among which two are typical medical imaging datasets (see Figures 3.12(a) and 3.12(c)) and one is a scientific visualization example (see Figure 3.12(b)). Our test system is a PC with an Intel ®XeonTM processor running at 2.66 GHz and a graphics card based on an NVidia ®GeForceTMFX 6800 chipset. Our implementation is cross-platform, having being tested both in a Windows and in a Linux operating system. The timing data presented here correspond to the Linux tests.

The first dataset (see Figure 3.12(a)) is a magnetic resonance (MRI) scan of a human head with a resolution of $0.90 \times 0.90 \times 1.09$ mm and a size of $256 \times 256 \times 114$ voxels. For this volume, we have extracted the isosurface corresponding to the isovalue 190 using *Marching Cubes*. The second dataset (see Figure 3.12(b)) is the result of a simulation representing the spatial probability distribution of electrons in a high potential protein molecule with a volume size of $64 \times 64 \times 64$ voxels. The represented isosurface corresponds to the isovalue 35. Finally, the third dataset is a computed tomography (CT) scan of a human chest obtained at a resolution of $0.71 \times 0.71 \times 1.25$ mm and a size of $512 \times 512 \times 294$ voxels. Here, the *Marching Cubes* algorithm is employed to extract the isosurface corresponding to the isovalue 115. All datasets are stored with an internal resolution of 8 bits/voxel.

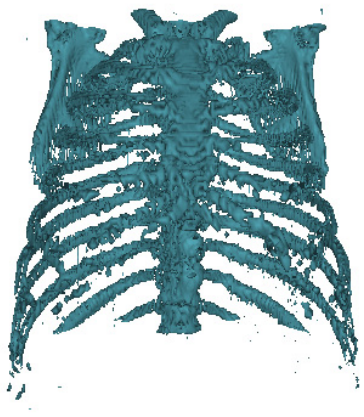
The obtained isosurfaces have been reconverted in each case into a sequence of triangle strips, in order to minimize the amount of connectivity information. The vertices of these meshes have then been encoded following the method described in Section 3.3.2. For the evaluation of the algorithm, the performance obtained with and without encoding are compared. When rendering the mesh without encoding, the standard OpenGL pipeline is utilized. For the encoded version, the geometry is uploaded as the content of a set of textures and decoded and rendered on-the-fly on the GPU. In order



(a) MRI Head



(b) Neghip



(c) CT Thorax

Figure 3.12: Isosurfaces utilized to evaluate the proposed method [6].

to obtain a representative value for the time required to upload the encoded geometry to the graphics card, an average value over 50 uploads has been measured. For the obtention of significative timing values of the rendering, the same benchmarking sequence has been used in all the tests. Table 3.1 summarizes the information collected during the performed tests.

Table 3.1: Timing results obtained for test isosurfaces. T_{OpenGL}: rendering time using standard OpenGL. T_{GPU}: time for decoding and rendering using our GPU-based method. T_{Upload}: extract from T_{GPU} employed to upload the encoded isosurface.

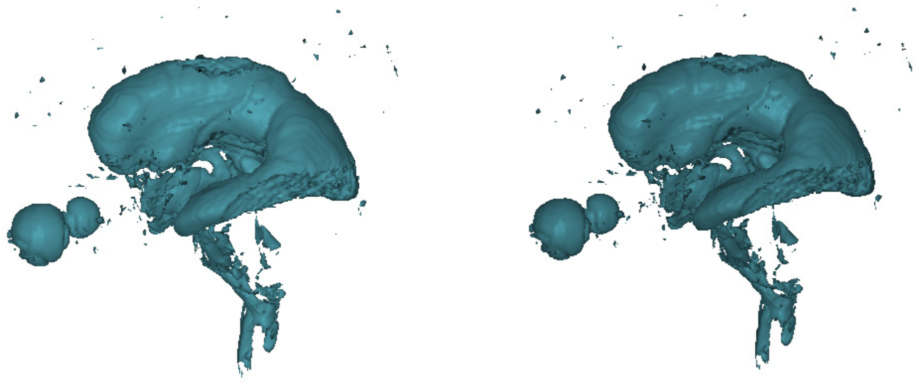
	MRI Head	*Neghip*	*CT Thorax*
Volume Size (#Voxels)	$256 \times 256 \times 110$	$64 \times 64 \times 64$	$512 \times 512 \times 294$
Isosurface Size (#Strips)	26401	6284	61566
Isosurface Size (#Verts.)	69924	17183	144933
T_{OpenGL} (ms)	42	11	88
T_{GPU} (ms)	22	5	41
T_{Upload} (ms)	3	1	4
FPS_{OpenGL}	24	91	11
FPS_{GPU}	45	200	24

As can be seen in the table, acceleration factors between 1.9× and 2.2× are achieved when the GPU-based decoding and rendering approach is utilized. This speed-up is mostly due to two different factors: first, the reduced size of the encoded vertices reduces the time necessary to upload the mesh from the CPU to the GPU compared to the standard OpenGL implementation; and second, since all the geometry is decoded and stored directly in graphics memory, it is already on the graphics card and ready to be rendered, thus making unnecessary any additional transfer from the CPU to the GPU in subsequent frames. While a similar benefit to that derived from the latter factor can be achieved by caching the whole mesh in the graphics card using OpenGL *vertex buffer objects (VBO)*, the former is a direct consequence of employing a purely GPU-based method. Furthermore, it should be noted that, even though these tests have been performed on static isosurfaces, our algorithm is perfectly suitable for accelerating the rendering of dynamic isosurface representations. The visualization of dynamic datasets, with time evolving inner structures, or isosurfaces with a changing isovalue, implies a frequent upload of the isosurface which in the extreme case could be every frame. In such a situation, the ability to render a compressed version of the isosurface acquires an extra value, as the traffic exchange on the bus between CPU and GPU usually becomes the major bottleneck for the rendering performance. As the results displayed on the table clearly show, the time required to upload the textures comprising the encoded isosurface is rather short, representing only the 10% of the total rendering time for the largest test dataset (CT Thorax). This would trivially result in a much higher acceleration factor than the one obtained for static scenes.

The main contribution of this new method, the capability of dealing with compressed representations of an isosurface, decompressing and rendering fully on the GPU without any further communication back to the CPU, opens a new door to the application of many compression schemes to real time visualization problems. While the encoding scheme employed in our proof-of-concept implementation is rather simple, the utilization of more complex compression algorithms, such as entropy based encoding, might be perfectly possible based on the same procedure as we have described here. An example of high performance rendering of compressed volume data using wavelets compression and direct volume rendering has been presented by Guthe et al. [53].

Furthermore, the framework and special utilization of available extensions one step further than the purpose they were created for, that have been performed for this project open many new possibilities for GPU-accelerated solutions which were not possible before. A good example of such possibilities is the work by Shiue et al., developed simultaneously to our solution for isosurfaces, and presented at the 2005 Siggraph [116], which describes a similar proposal applied to subdivision on the GPU. Moreover, other applications such as displacement mapping, may now also be produced on the GPU without the necessity of extra slow transfers between the CPU and the graphics pipeline.

Concerning the image quality achieved with our method, Figure 3.13 presents both rendering results corresponding to rendering the isosurface with standard OpenGL and to applying our decoding and rendering approach running fully on the GPU. As the choice of the resolution for the internal representation of the encoded values has been performed taking into account the limitations of the hardware (e.g., internal precision on the GPU), both images reproduce equivalent visual accuracy. It must be noted at this point that, due to the fact that the GPU-based implementation carries out all lighting effects directly in the shader program responsible for rendering the mesh, slight differences in the illumination conditions can be appreciated in both images.



(a) Original (b) Decoded

Figure 3.13: Image quality comparison [6].

3.4 Summary

In this chapter, the application of indirect volume rendering techniques has been addressed. Given the characteristics of most medical scanned data, such as regularity or their scalar structure, the extraction of isosurfaces and their subsequent display represent a very important role in medical visualization. Therefore, both aspects, the isosurface extraction process and the succeeding rendering of those isosurfaces have been analyzed. First, the standard method for isosurface extraction, the *Marching Cubes* algorithm, has been presented, followed by a survey of the most relevant optimization techniques oriented to accelerate the computation of the polygonal model representing the sought after isosurfaces. Once the current state-of-the-art has been settled, a review of compression techniques applied to the visualization of isosurfaces obtained out of scalar volumes has been performed. Our own contribution, a novel method for accelerating the rendering of large isosurfaces based on the utilization of the programmability of current graphics cards, has then been presented in detail. Our strategy works on an encoded version of the isosurface that reduces its memory size, while other encoding and compression schemes are compatible with the method and might be incorporated in the future. The main contribution of our work focuses on decoding and rendering the isosurface. By making use of the programmability of recent graphics cards and the newly available frame-buffer objects OpenGL extension, the whole decoding and rendering process can be performed directly on the GPU. Since no slow read back to main memory through the CPU is required, the rendering performance is increased. One major bottleneck for the rendering of large isosurfaces, the bandwidth of the bus connecting CPU and GPU, is also alleviated due to the reduced size of the encoded isosurface. This can greatly benefit applications where the isosurface must be updated very often, like the visualization of time sequences of animated volumes. The obtained results are encouraging and illustrate the benefits of our method: reduced bandwidth requirements and faster quality rendering.

As a possible line for future work, the investigation of the feasibility of implementing a GPU-accelerated entropy decoder is a very promising avenue and a perfect complement to the procedure described in this chapter. Given the ability to *generate* geometry directly on the GPU, as it has been proven in our implementation, derived from the utilization of the newly available OpenGL *frame-buffer objects*, a whole new generation of GPU-accelerated algorithms can be now developed. Such techniques include, but are not limited to, GPU based subdivision, GPU based displacement mapping, etc.

In the last two chapters, several techniques for the visualization of volume datasets have been presented and analyzed, either utilizing point-based rendering (Chapter 2), or indirect volume rendering (Chapter 3). In both cases, the focus has been set on acceleration techniques devoted to speed up the rendering process while minimizing or even avoiding visual quality losses. In the following chapter, a rather different perspective is adopted for direct volume rendering applied to medical visualization. Now the target will not be the reduction of the rendering time, but to improve the quality and meaningfulness of the visualization experience. Therefore a special emphasis will be put on the classification stage of the direct volume rendering pipeline and on mechanisms for assisting the user in order to efficiently display the relevant information out of the volume data.

CHAPTER 4

Direct Volume Rendering

4.1 Introduction

In contrast to indirect volume rendering, where the volumetric data are first converted into a set of polygonal isosurfaces and subsequently rendered with standard polygonal graphics hardware, with direct volume rendering the data are directly rendered without any intermediate conversion step. This has the advantage of providing a complete insight of the data as a whole, as the complete 3D sampled volume is represented in the final rendered image. Furthermore, by working with the whole volume, it is possible to explore not only the external surface as with indirect volume rendering, but also the inner structures hidden underneath those surfaces. Such characteristics make volume rendering[1]of great interest as a supporting tool in medical scenarios by both improving the global understanding and providing a complete three-dimensional representation of the scanned patient's data. Of course such insight could be also achieved by means of extracting and visualizing a set of N different isosurfaces, but at the cost of much higher memory and processing requirements, which make this alternative not suitable for general purposes.

It is important to note that in volume rendering, unlike other areas of computer graphics, photo-realism is not the goal. In fact, given the nature of the data being visualized (e.g., scanned data), in many situations it is not even possible to define what a photo-realistic result should look like. So, the task to be achieved with volume rendering is to synthesize meaningful images from 3D data such that the resulting rendering reveals useful insights to the user.

Many different algorithms have been designed as a realization of the direct volume rendering concept. Among these, four techniques have become the most popular, thus establishing a standard set for volume rendering: *ray casting* [124, 74], *splatting* [131], *shear-warp* [71], and *3D texture mapping* [31]. All these algorithms are based on the *direct volume rendering integral*, also known as the *low-albedo volume rendering integral*, which is described in Section 4.1.1. Independently of the chosen algorithm, volume rendering is always achieved as a sequence of ordered operations forming the so-called *volume rendering pipeline*, which is briefly summarized in Section 4.1.2. Subsequently, Section 4.1.3 presents the basics of the four most widespread volume rendering algorithms mentioned above, hence completing this short reminder of the essentials of volume rendering. Then, in Section 4.2, special emphasis is put on the classification stage of the volume rendering pipeline. Since a proper classification is a crucial element for the obtention of meaningful and practically valuable images, its adequate design plays a fundamental role in the utilization of volume rendering as a medical visualization tool. The relevance of this process is clearly stated in Section 4.2.1, where our original proposal for transfer function definition is presented in detail.

While the original idea, design of the algorithm, measurement and analysis of the results, as well as the whole final presentation have been produced by the author of this thesis, the prototype implementation utilized for the evaluation of the proposed method was performed by Martin Köbele as a part of his master's project (*Diplomarbeit*) [67] under the active supervision of this author. It should also be acknowledge the participation of Jan Fischer as responsible for the medical AR environment and fruitful discussions.

[1]The term *direct volume rendering* is commonly used as opposed to *indirect volume rendering*. The shorter form *volume rendering* is usually employed as a synonym of *direct volume rendering*. In the remainder of this chapter, for the sake of simplicity, the former will be often used and should not be mistaken with indirect volume rendering.

4.1.1 Volume Rendering Integral

The volume rendering integral (see Equation 4.1) is the solution to an equation computing the color of light that passes through a volume. Its theoretical basis is the *density emitter model* introduced by Sabella [109], which assumes a simplified model of the transport theory of light and its interaction with the matter being visualized. This simplified model takes only absorption and emission into account, hence ignoring other physical terms such as scattering or refraction. Such simplification is usually known as the *low albedo* model, making reference to the low reflectivity assumed for the particles forming the volume, which leads to neglecting the effects of inter-reflection within the volume [74, 109, 43].

$$I_\lambda(x,\vec{r}) = \int_0^L C_\lambda(s)\ \mu(s)\ e^{-\int_0^s \mu(t)\,dt}\,ds \tag{4.1}$$

Therefore, $I_\lambda(x,\vec{r})$ in Equation 4.1 is the analytical solution to the amount of light of wavelength λ coming from ray direction $\vec{r}$ that is received by an observer at location x on the image plane. Here, L represents the length of ray $\vec{r}$. The terms in the body of the integral can be easily interpreted. The model assumes thinking of the volume as a continuum of particles with certain densities, μ, able to absorb and/or emit light. C_λ accounts for the amount of light of wavelength λ being reflected and/or emitted at a location s in the direction towards the observer, i.e., the direction of the ray $\vec{r}$. As the reflectivity associated to each infinitesimal location is proportional to the density of the *participating medium* (i.e., the volume) at that position, the reflected color (light at wavelength λ) is weighted by the particle's density, μ. The decreasing exponential function represents the attenuation of the light from position s, due to the density of particles between s and the observer's eye.

Unfortunately, even with this simplified model, in the general case the volume rendering integral (Equation 4.1) cannot be solved analytically [87]. Thus, in order to obtain a numerical solution, practical volume rendering algorithms perform a discretization of the integral by decomposing it into a Riemann sum, i.e. a series of sequential intervals i of width Δs along the ray $\vec{r}$, as expressed in Equation 4.2.

$$I_\lambda(x,\vec{r}) \simeq \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\ \mu(s_i)\ \Delta s\ \prod_{j=0}^{i-1} e^{(-\mu(s_j)\ \Delta s)} \tag{4.2}$$

Now, the exponential term can be replaced by its Taylor series, $e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!}$. Furthermore, the full series expansion can be truncated dropping all but the first two terms (i.e., linear approximation). Performing these substitutions gives place to the *discretized volume rendering integral* presented in Equation 4.3, where $\alpha(s_i) = 1.0 - \tau(s_i)$ is the opacity at interval i (τ is the transparency).

$$I_\lambda(x, \vec{r}) \simeq \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\ \alpha(s_i) \prod_{j=0}^{i-1} [1 - \alpha(s_j)] \tag{4.3}$$

It must be noted that, due to the non linear behavior of the discretized volume rendering integral, the composition must be performed in sorted order, either front-to-back, or back-to-front.

Definition 4.1.1 Front-To-Back Composition

For each pixel on the screen, x, the intensity for each color component corresponding to the front-to-back composition of all voxels along a direction $\vec{r}$, can be computed as

$$I(x, \vec{r}) = \sum_{i=0}^{n} C_i\ \alpha_i \prod_{j=0}^{i-1} (1 - \alpha_j)$$

This can be expressed iteratively as

$$\begin{aligned} C_{out} &= C_{in} + (1 - \alpha_{in})\ \alpha_i\ C_i \\ \alpha_{out} &= \alpha_{in} + \alpha_i\ (1 - \alpha_{in}) \end{aligned}$$

In both definitions (Definition 4.1.1 and 4.1.2), C_{out} and α_{out} are the total accumulated color intensity and opacity just after the composition of the current sample point; C_{in} and α_{in} are the total accumulated values just before compositing the current sample point; and C_i and α_i are the color intensity and opacity of the current sample point.

4.1.2 Volume Rendering Pipeline

Every direct volume rendering algorithm is achieved as a series of concatenated steps. As each operation in the sequence is completed, the data is passed to the next operation, forming a *pipeline*. The general structure of this

Definition 4.1.2 Back-To-Front Composition

For each pixel on the screen, x, the intensity for each color component corresponding to the back-to-front composition of all voxels along a direction $\vec{r}$, can be computed as

$$I(x, \vec{r}) = \sum_{i=0}^{n} C_i \ \alpha_i \prod_{j=i+1}^{n} (1 - \alpha_j)$$

This can be expressed iteratively as

$$C_{out} = C_i \ \alpha_i + C_{in} \ (1 - \alpha_i)$$

---

pipeline is illustrated by Figure 4.1. The operations in the pipeline consist of segmentation, gradient computation, resampling, classification, shading, and compositing. The order and inclusion of some of these steps may vary among different algorithms and practical implementations. The consequences associated to some of these changes, such as the inversion of the relative order between resampling and classification, will be briefly analyzed below. But first, let us comment the meaning of each of the pipeline stages:

- **Segmentation:** It is a preprocessing step, typically done before the actual rendering. It consists of separating the dataset into structural units by labelling voxels within the volume. Each label identifies one feature and this information can be further utilized for separating these semantic classes during rendering. A clear example in medical imaging is the identification of bones and tissue regions from a patient's scan. The segmented regions can then, for instance, be assigned a different transfer function during classification in order to be highlighted and presented to the user.

- **Gradient Computation:** The gradient is a measure of how quickly voxel intensity in the dataset changes. The gradient also indicates the direction of this variation. Since the changes in intensity are normally perpendicular to the interface between two materials, the gradient vector is utilized as an approximation of the normal vector at a material's or object's surface. This information is typically used for shading, as in any traditional computer graphics discipline, but also can be extremely useful for the identification of material boundaries and edges. Several filters can be employed for the computation of the gradient,

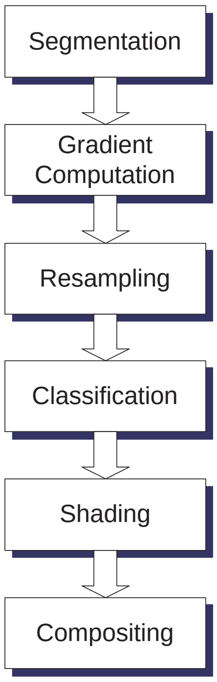


Figure 4.1: Direct volume rendering pipeline.

being the *central differences* operator the most simple and commonly applied one. Other possibilities such as the Sobel operator or cubic spline filters can provide a more accurate estimation of the gradient vector, but at the cost of being more time and memory consuming. A comparison of different gradient estimation techniques can be found in [27].

- **Resampling:** In volume rendering, voxel intensities along a viewing direction must be accumulated in order to produce a 2D image of the 3D dataset. Since this is a discrete process where arbitrary positions within the dataset must be accessed, the volume must be sampled. It is called resampling, as the volume is already a discrete sampling of the continuous patient's anatomy. Such resampling is necessary because these 3D arbitrary positions rarely coincide with an exact voxel location on the volumetric grid, thus requiring an interpolation based on the values at neighboring grid positions. There exist many approaches to perform this interpolation, each one controlled by a different interpolation kernel. Usually separable kernels are utilized, since they allow the interpolation being performed independently in each dimension. The most straightforward method is nearest neighbor interpolation, where the voxel closest to the sought after position is selected. Even though this is easy to implement and rather fast, nearest neighbor interpolation does not provide high quality results. A more accurate and very common alternative is the utilization of trilinear interpolation within a volume cell. Using trilinear interpolation, the eight voxels forming the cell containing the query position are utilized to compute this location's value. Linear interpolation is then consecutively applied along each of the three coordinate directions. Despite its relatively high complexity, due to the optimizations suffered by modern graphics hardware (e.g., 3D texture memory), good performance can be achieved. Its major drawback is the appearance of diamond-like artifacts due to the nature of the trilinear kernel. As with the case of gradient computations, more complex interpolation kernels (e.g., Gaussian kernel, cubic convolution, B-spline interpolation) may be utilized at the cost of longer computing times and larger memory requirements, should the accuracy provided by trilinear interpolation not be enough.

- **Classification:** Decisive step in every volume rendering algorithm, classification consists in the assignment of optical properties such as color and opacity to every voxel in the volume. This assignment is based on the application of a transfer function which defines the mapping between internal parameters of the 3D data (e.g., intensity, gradient magnitude, curvature) and the final color and opacity utilized during rendering. A detailed description of the classification stage and several algorithms for the design of transfer functions are presented in Section 4.2.

- **Shading:** Illumination and shading within volume rendering borrow the concepts and techniques common to traditional polygon-based rendering. By simulating the effects of light-matter interaction, the appearance of rendered objects is enhanced. This interaction is typically factorized into three modelling elements: ambient, diffuse, and specular light components. The ambient component is present at each position in the scene as a constant distribution that approximates the effect of inter-reflections. The diffuse component models the interaction of light with rough materials, where light scatters almost equally in all directions. Therefore it depends on the normal vector but not on the viewpoint. Finally, the specular component depends on the angle between the light and the eye position and specifies how much of a light source's intensity is reflected, thus simulating the behavior of smooth, shiny objects.

Two are the main differences between shading in polygon-based rendering and volume rendering: The first difference is trivial: while in polygon-based rendering these effects are applied to surface elements (i.e. triangles), with volume rendering, volumetric objects are utilized instead. The second factor consists in the goal to be achieved in both cases. Differently to most polygon-based rendering examples, by applying illumination and shading effects to volume rendering no photorealism is pursued. On the contrary, the intention is to provide a better understanding of the inner structures of the volume and thus achieve a more meaningful representation of the 3D data.

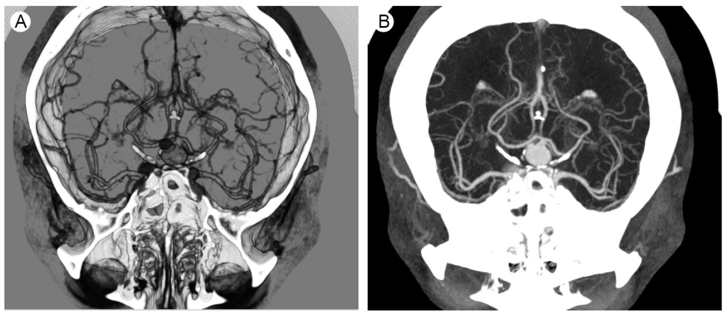


Figure 4.2: Comparison of (A) direct volume rendering compositing and (B) *maximum intensity projection* (MIP) [44].

- **Compositing:** In the last stage of the pipeline, all contributions to each pixel in the final image are combined into one color value. This process represents an approximation to the *discrete volume rendering integral* presented in Equation 4.3. Depending on the chosen algorithm, this gathering process for the corresponding color and opacity values can be performed either in front-to-back or back-to-front order. A survey of various volume rendering algorithms is provided in Section 4.1.3. Other compositing operators can be chosen instead of the accumulation over all samples to determine the color for each pixel using the volume rendering integral. One example of such operators is known as *maximum intensity projection (MIP)*. It selects the maximum voxel intensity of all samples along every viewing direction, hence producing results resembling an X-ray image of the dataset. Maximum intensity projection compositing is commonly used in medical imaging to highlight high-contrast features within an MRI scan, such as arteries in rotational angiography acquisitions [2]. Figure 4.2 shows a comparison

[2]In these cases, arteries appear as bright structures due to an injected contrast agent.

of direct volume rendering compositing and MIP applied to the same dataset.

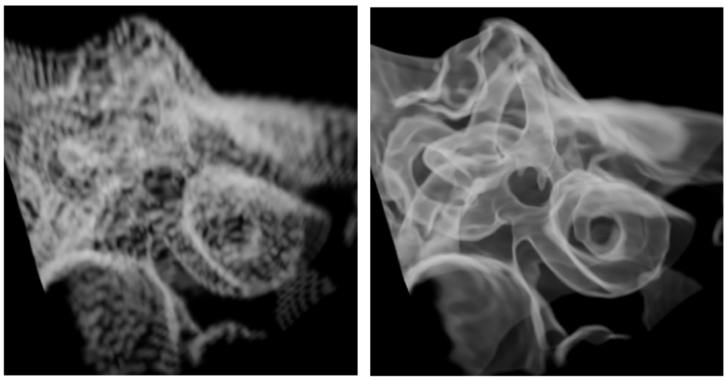


Figure 4.3: Pre-classification (left) versus post-classification (right) [44].

As briefly mentioned above, the position of some of the elements in the volume rendering pipeline can be exchanged. Specially important are the consequences derived from the relative order between *resampling* and *classification*. Generally, applying the transfer function responsible of defining the classification of the volume before the interpolation performed for resampling has taken place can cause visible artifacts. This approach is usually referred to as pre-classification. When using pre-classification, a colored volume is generated where each voxel is represented by an RGBα value instead of the scalar intensity typical of medical datasets. Therefore, by executing classification first, the interpolation takes place in the 4D color space, leading to artifacts such as image blurring and *color bleeding* [135].

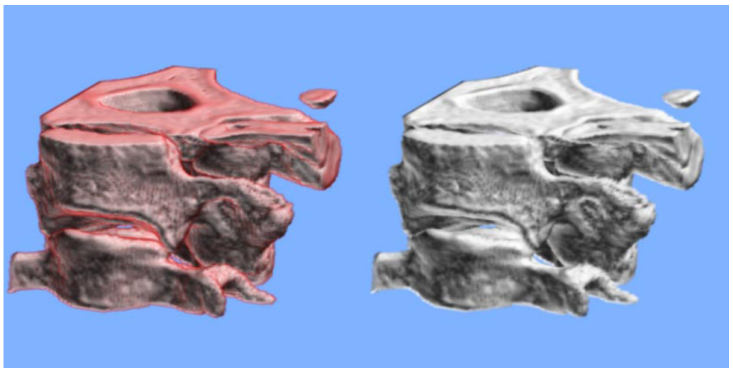


Figure 4.4: Color bleeding example. Separate interpolation of color and opacity (left) versus opacity-weighted interpolation of colors (right) [135].

These problems can be minimized if opacity-weighted color interpolation is employed instead of separately interpolating color and opacity values. Otherwise, low-opacity colors may mix on equal terms with high-opacity colors, leading to color bleeding artifacts at the boundaries between differently colored regions of the volume as illustrated by Figure 4.4.

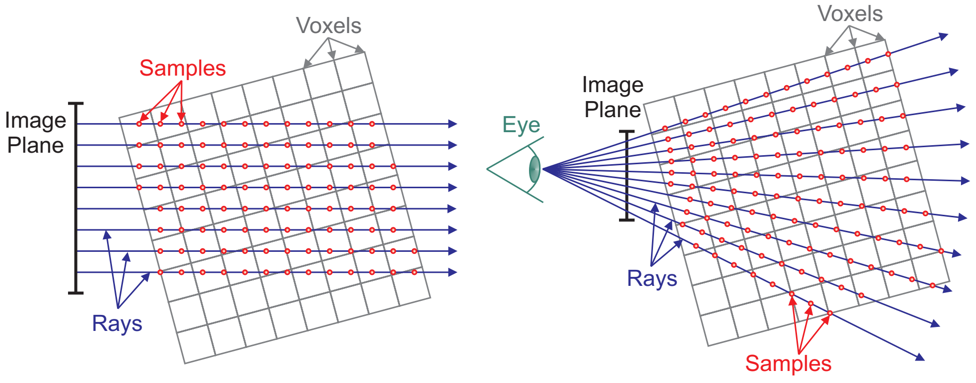
Such effects are obviously avoided with post-classification, where interpolation takes place in the 1D voxel intensity space and the classification is performed on these already resampled values. However, it should be noted that with post-classification artifacts might be generated too. In particular, due to the interpolation of intensity values, if a material with a high voxel value is adjacent to a material with a low voxel value, the averaging introduced by interpolation can visibly distort the obtained voxel values. This way, one might obtain interpolated values being classified as a structure that is not present at the given location. This is known as the *partial volume effect* and, together with aliasing artifacts, is one of the main problems associated with undersampling.

4.1.3 Volume Rendering Algorithms

Many are the algorithms devoted to create an image from of a volumetric dataset by implementing the discretized volume rendering integral presented in Equation 4.3. In this section a brief survey is presented of the main characteristics of those techniques that have become the standard set for direct volume rendering.

4.1.3.1 Ray Casting

Ray casting [74] is an image-order direct volume rendering algorithm performing a straightforward numerical evaluation of the volume rendering integral. Its working principle is therefore quite simple: for each pixel of the image, a ray is cast from the viewpoint into the scene traversing through the volume. For each of these rays, samples are calculated at given locations along the ray path in order to approximate the integration process. Typically, equally-spaced samples are used for the sampling, while optimizations might affect such regular sampling for the sake of performance and/or quality. A reconstruction filter is utilized for interpolating the discrete volume and computing the corresponding value at each sample location. As trilinear interpolation is the most frequently selected reconstruction, the scalar values of eight neighboring voxels are weighted according to their distance to the actual location for which a value is needed. A transfer function is then applied to properly classify the volume, if post-classification is used. Pre-classification may also be applied instead, by inverting the order between interpolation and classification, as already discussed in the previous section. The solution of the volume rendering integral is then approximated by compositing the obtained sample values via alpha-blending in either back-to-front or front-to-back order.



(a) Orthogonal Projection (b) Perspective Projection

Figure 4.5: Ray casting of a volume dataset with uniform sampling.

Figure 4.5 illustrates the compositing process in a typical ray casting scenario for a front-to-back implementation for both orthographic (Figure 4.5(a)) and perspective projection (Figure 4.5(b)). The samples along each ray are subsequently combined as a weighted sum controlled by the corresponding opacity (alpha) value, thus producing the rendered image. It must be noted that, in general, the highest frequency of the content of the volume is unknown, and therefore the Nyquist theorem[3] must be taken into account to determine the sampling frequency (i.e., the distance between consecutive samples along a ray).

Theorem 4.1.1 Nyquist Theorem

When sampling a band-limited signal, the sampling frequency must be greater than twice the input signal bandwidth in order to be able to reconstruct the original perfectly from the sampled version.

[3]The Nyquist theorem is also known as the Shannon theorem or just simply as the sampling theorem.

In this case, this implies that the spacing between two consecutive samples should be smaller than half of the voxel grid spacing.

As the ray casting algorithm is a rather computing intensive technique, acceleration methods are usually applied to increase the rendering performance. When using front-to-back compositing, *early ray termination* may be employed to accelerate the sampling process. As its name suggests, the sampling along a ray is stopped once full (or almost full) opacity has been reached ($\alpha \approx 1.0$), since the structures behind that sample location would not contribute to the final image. Another useful acceleration technique is *empty space leaping*, where a distance field or an equivalent data structure is applied to identify and avoid empty regions where no sampling is required.

Despite these optimizations, the performance obtained with software implementations of ray casting is usually not enough for real-time or even interactive frame rates. Therefore, specific hardware solutions such as the *Vizard II* [89, 90, 62] or the *VolumePro* [97] boards have been developed, which provide a significant boost in terms of rendering speed without sacrificing image quality. Additionally, in recent years, due to the programmability of modern graphics cards, an intermediate solution has arisen in the form of GPU-based ray casting implementations [70, 106]. These approaches rely on the new Pixel Shader 2.0 model (DirectX) or the equivalent functionality via OpenGL shader programs. Describing the details about these techniques is clearly beyond the scope of this thesis, but a complete survey of the key factors to be taken into account in GPU-based ray casting is presented in the *2004 SIGGRAPH Course Notes* by Engel et al. [44].

4.1.3.2 Splatting

The splatting algorithm was first proposed by Westover [131]. In contrast to ray casting, splatting is an object-space oriented volume rendering algorithm. In splatting, each voxel is represented by overlapping radially symmetric basis functions which are pre-integrated into a 2D footprint, weighted by the voxel value and mapped onto the image plane. A Gaussian kernel is commonly selected as the basis interpolation function. The pre-integrated 2D footprint is stored in a lookup table, hence reducing the rendering time.

In the original splatting algorithm, all splats are composited back-to-front, which is prone to produce bleeding artifacts from hidden objects. A straightforward optimization of this technique consists of locally accumulating slices of splats in cache-sheets, aligned with the volume face most parallel to the image plane. This introduces inaccuracies that result in severe bright-

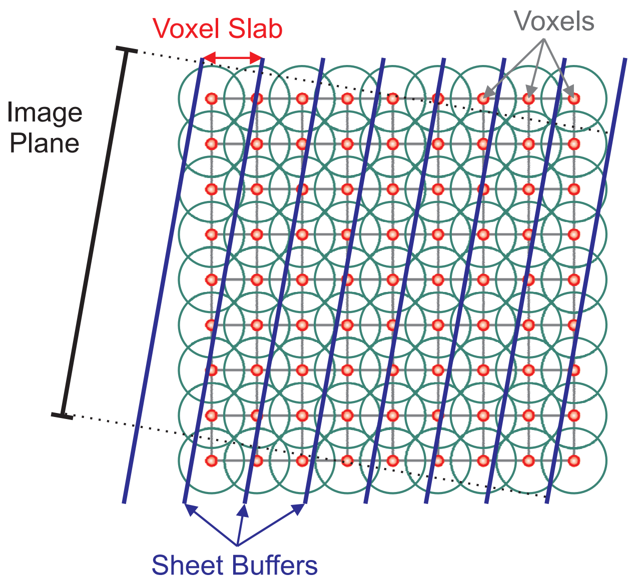


Figure 4.6: Splatting. Illustration of image aligned splatting.

ness variations, such as visible popping artifacts in animated views. Image aligned splatting [92], eliminates most of these drawbacks by projecting slabs of the voxel kernels onto *sheet buffers* of a constant width Δs, aligned parallel to the image plane (see Figure 4.6). Additionally, optimizations like the *early splat elimination* [93] of non-contributing splats can help to reduce the rendering time by reducing the amount of redundant computations. In a similar manner to the early ray termination for the ray casting algorithm, the opacity of the set of pixels covered by a splat is tested against a given threshold value. If the obtained opacity is already above the threshold, the contribution of the respective splat (and of the upcoming ones behind it) can be neglected and the front-to-back compositing can be interrupted.

It also must be noted that the splatting algorithm replaces the point samples of ray casting with a sample average across a distance given by the width of the slabs, Δs. This introduces additional low-pass filtering, if $\Delta s > 1.0$. Even though this helps to reduce aliasing, it also tends to smooth details in the volume. Furthermore, if post-classification is applied, the filtering of the original intensities tends to increase the partial volume effect. On the other hand, the size of the splats is a key factor for the performance of the algorithm. Therefore, a tradeoff between performance and visual quality is commonly necessary. This is one of the reasons for splatting usually not providing such a high image quality as ray casting. This, together with the increasing hardware support for texture mapping has lead to a certain loss of popularity of the splatting algorithm for direct volume rendering. Nevertheless, the splatting principle has become a powerful tool for point rendering applications, as has been presented in Chapter 2.

4.1.3.3 Shear-Warp

The shear-warp factorization [71] is a hybrid alternative between an image-space oriented (e.g., ray casting) and an object-space oriented (e.g., splatting) algorithm. It is mainly a variation of ray casting, which factorizes the viewing transformation into a shearing and a warping step. In contrast to ray casting, no rays are cast back into the volume, but the volume itself is projected slice by slice onto the image plane. The data reconstruction is performed as bilinear interpolation within two-dimensional slices, instead of the trilinear interpolation typically used by ray casting.

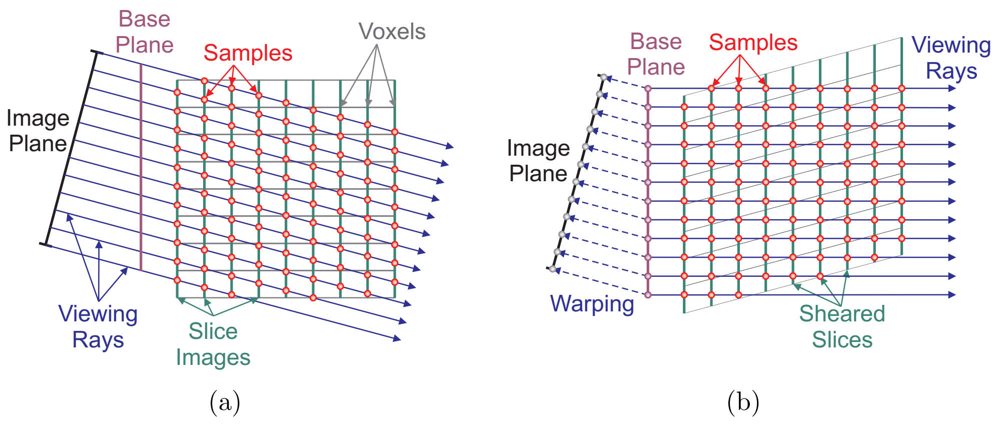


Figure 4.7: Shear-warp factorization for orthogonal projection.

The basic principle of shear-warp is illustrated in Figure 4.7 for the case of orthogonal projection. The projection does not take place directly on the final image plane, as with ray casting, but on an intermediate image plane called *base plane*. The base plane is located in front of and aligned parallel to the volume instead of the viewport. Then, the volume itself is *sheared* in such a way that the viewing rays become perpendicular to the volume slices (see Figure 4.7(b)), which allows for an extremely fast implementation of the projection due to the optimal memory access. The rays obtain their sample values using bilinear interpolation within the traversed volume slices. This results in a view-dependent sampling interval, which can vary between 1.0 for axis-aligned views, to $\sqrt{3}$ for corner-on views. Thus, the Nyquist theorem is potentially violated, which can result in visible aliasing artifacts.

In the second step of the algorithm, the computed base plane image is *warped* onto the final image plane, as illustrated in Figure 4.7(b). Note that this is only necessary once per image and not per slice. The example shown in Figure 4.7 corresponds to orthogonal projection. Perspective projection can be easily incorporated to the algorithm by scaling the volume slices in addition to the shearing in order to address the divergence of perspectively cast rays.

The set formed by the above mentioned characteristics, together with additional optimizations such as run-length encoding the volume data, is what make the shear-warp algorithm probably the fastest software method for direct volume rendering.

4.1.3.4 Texture Mapping

This object-space oriented algorithm is one of the most commonly applied volume rendering techniques. Two are the main existing variants of texture mapping applied to volume rendering: 2D and 3D texture mapping. Both techniques consist of an accumulation of parallel texture slices in a back-to-front manner using the (alpha-)blending functionality of graphics hardware. The main difference is the type of alignment applied in each case, which in turn results also in a different interpolation scheme.

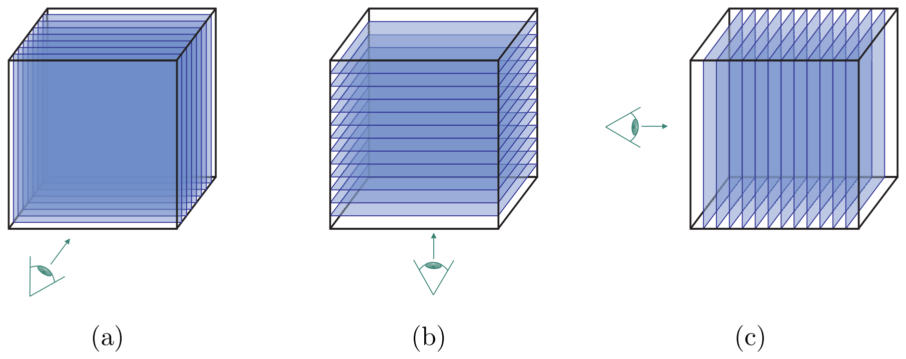


Figure 4.8: 2D texture mapping-based volume rendering with object-aligned textures.

In 2D texture mapping [31], object aligned slices are alpha-blended to produce the final image. Three stacks of planar slices aligned with each of the three major axes of the volume are utilized as proxy geometry. This alignment is necessary so that the volume can be mapped with 2D textures, which are then resampled using bilinear interpolation within each slice. During rendering, the stack corresponding to the major axis which is most parallel to the viewing direction is chosen (see Figure 4.8). This may lead to visible popping artifacts when switching between two major axis directions. Furthermore, as only 2D textures are employed, no interpolation between slices is performed, thus limiting the final image quality due to under-sampling along the view direction. In addition, the necessity of creating and keeping three stacks of object-aligned slices clearly increases the texture memory requirements. On the other hand, by relying only on 2D texture memory, this type of implementation is directly supported by any graphics board. Also, the rendering performance is extremely high, since bilinear interpolation requires only a lookup and a weighting of four texels for each resampling operation.

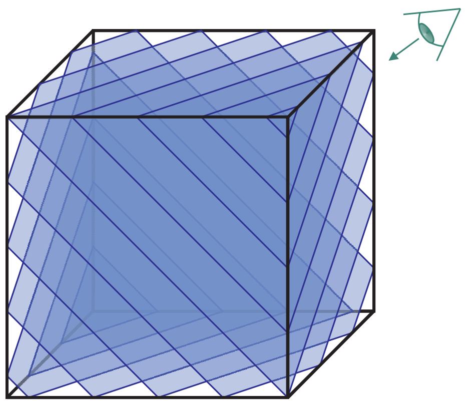


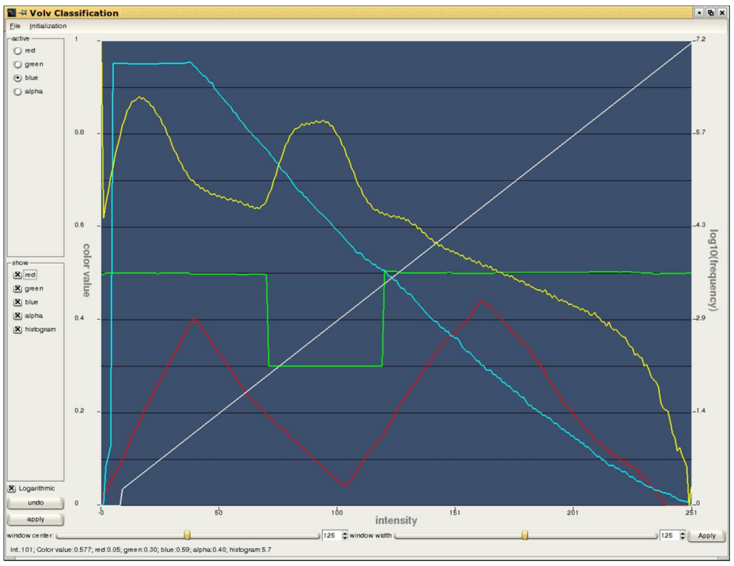
Figure 4.9: 3D texture mapping-based volume rendering with view-aligned textures.

Many modern graphics cards include support for 3D textures, additionally to traditional 2D textures, thus enabling the possibility to implement 3D texture mapping [38]. In this case, the dataset is stored in a single 3D texture map, which can be directly indexed in order to obtain a value at any arbitrary location within the volume by applying trilinear interpolation. Since no 2D restriction is bound to the hardware as with 2D texture mapping, the natural selection is to use slices aligned with the viewport (see Figure 4.9), as these more closely reproduce the sampling employed by ray casting. Furthermore, due to the hardware supported trilinear interpolation within the volume, proxy slices (i.e., polygons) are not limited to original slices from the dataset. This way, the number of slices and the sampling distance between them can be adjusted on-the-fly without any restrictions, so that a constant sampling rate for all pixels and viewing directions can be maintained. This not only eliminates the popping artifacts typical of 2D texture mapping, but also reduces aliasing problems associated with under-sampling.

On the other hand, three are the main drawbacks of 3D texture mapping: first, the still limited availability of 3D texturing capabilities in consumer's graphics cards; second, trilinear interpolation is significantly slower than bilinear interpolation, due to the necessity of accessing eight texels per sample (instead of four), and texture fetch patterns that decrease the efficiency of texture caches; and third, larger volumes which do not fit into texture memory require swapping of volume bricks between main memory and texture memory with the consequent performance drop. Nevertheless, in recent years, 3D texture mapping has become popular in PC based graphics hardware, which in turn has produced an increase of the on-board texture memory available, thus limiting the effect of the aforementioned hindrances. As a result, 3D texture mapping has become a standard solution for those visualization applications that do not require the accuracy of ray casting, but a good tradeoff between rendering performance and image quality.

4.2 Classification in Direct Volume Rendering

As an element of the direct volume rendering pipeline, classification is defined as “the process of identifying features of interest based on abstract data values” [44]. Normally, the classification step is performed applying a transfer function that maps the domain of data values from the original volume (e.g., voxel intensity, gradient, curvature) to the range of optical properties such as color and opacity. Finding a good transfer function for volume classification is one of the key problems in direct volume rendering, as a good classification is mandatory for a meaningful visualization. Despite its crucial relevance, it has not been until recent years that a considerable research effort has been put on addressing this problem [98]. In the simplest case, a transfer function has a one dimensional domain (voxel intensity), while its range can be characterized by one (opacity) or four dimensions (color and opacity). Typically, the user is presented with a transfer function editor working on a histogram basis, that visually demonstrates the effect of changes in the transfer function. An example of such an editor is presented in Figure 4.10, where the yellow line corresponds to the histogram of the volume dataset and a different mapping function is defined for the red, green, blue and alpha channels. This rather naive approach, even though still being used in many applications, leads to a complex and time consuming trial and error process. Obviously, the addition of new dimensions to the domain of the transfer function further complicates this explorative task.

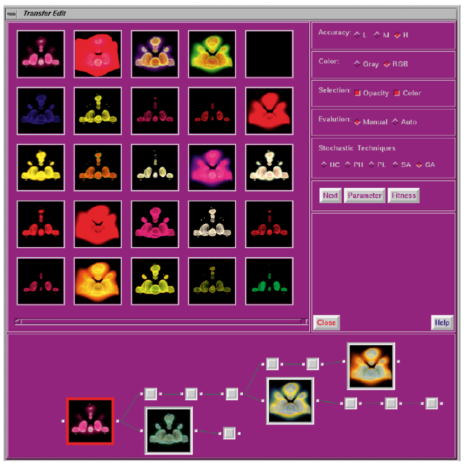


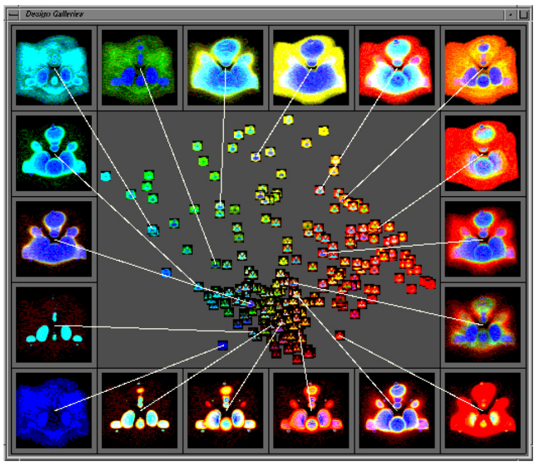
Int.:101; Color value:0.577; red:0.05; green:0.30; blue:0.59; alpha:0.40; histogram:5.7

Figure 4.10: Transfer function editor working on a 1D histogram basis.

In order to simplify and facilitate this cumbersome design process, various alternatives have been proposed. He et al. [54] treat the search for a proper one-dimensional transfer function as a stochastic parameter optimization problem and employ heuristic search techniques, either based on user selection of intermediate thumbnail renderings (see Figure 4.11(a)), or automatically controlled by user-specified objective image fitness functions. The purpose is to create an interface that suggests an appropriate transfer function and lets the user to guide the design process based on how well he/she feels the rendered images capture the essential features of the dataset.

Marks et al. [85] address the general problem of computer-assisted parameter setting with a Design Gallery interface. Their approach generates a selection of images obtained with a broad selection of all possible transfer functions. The whole space of transfer functions is parameterized and stochastically sampled, thus producing volume rendering images illustrating each parameter selection. The images are then grouped based on similarity. An example of such a user interface is provided in Figure 4.11(b). Since many different images must be rendered, this approach can be quite time consuming but it may be performed in a fully automatic manner. A similar


(a) Stochastic search [54]


(b) Design gallery [85]

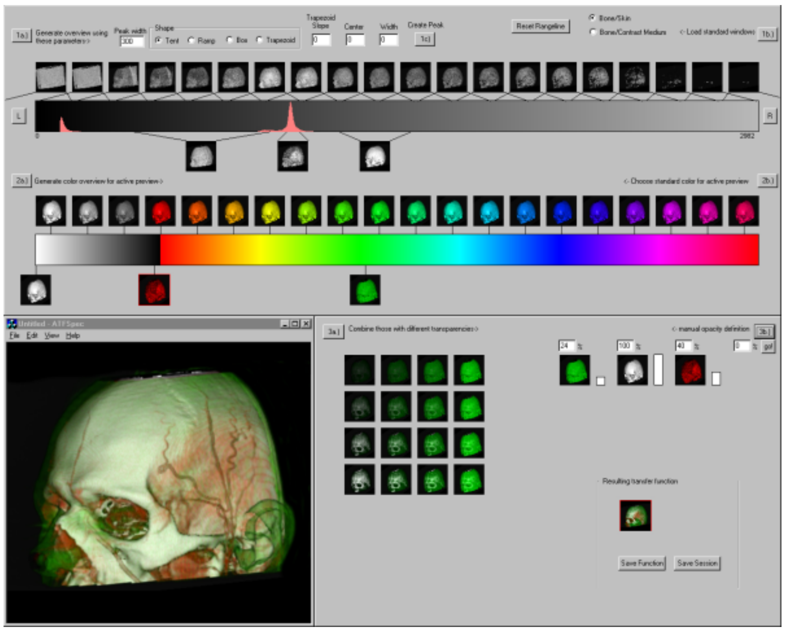

(c) Thumbnail parameterization [68]

Figure 4.11: Examples of transfer function design interfaces.

approach is presented in [68], where thumbnail renderings are also employed to assist the user, but decoupling the different search domains (data range, color, opacity) and benefiting from the use of the VolumePro technology [97] to provide real-time visual feedback (see Figure 4.11(c) for an illustrative example). The Contour Spectrum proposed by Bajaj et al. [25] tackles the problem in a more data-centric manner. Data metrics such as surface area and mean gradient magnitude are utilized to parameterize and visually summarize the space of isosurfaces in the volume. Even though this is devoted to guide the choice of isovalues for indirect volume rendering, the provided information can also be useful for transfer function generation.

The above mentioned techniques propose alternatives to facilitate the design of one-dimensional transfer functions. However, over the years it has become clear that the use of multi-dimensional transfer functions can drastically benefit the success of volume visualization [98]. Such approaches, already proposed by Levoy in 1988 [74], incorporate additional parameters to the domain of transfer functions, hence increasing their capability to visually discern between different materials and structures. Such functions can be defined as a mapping from 2D to 1D,

$$f : \mathbb{R}^2 \mapsto \mathbb{R}$$

where the domain $D \subseteq \mathbb{R}^2$ is formed by the voxel's value (or intensity), I, and its gradient magnitude, $|\nabla I|$; while the function's range $R \subseteq \mathbb{R}$ defines the opacity value, α, for each voxel.

Kindlmann et al. [63] use first and second derivative information in the opacity transfer function design in order to semi-automatically isolate structures within the volumetric dataset correlating with a material boundary model. A different and interactive approach is presented in [65], where the gradient magnitude is also computed together with the Hessian matrix, but where the color is added to the range of the transfer function. This derivative information is thus incorporated into the transfer function design process by a set of manipulation widgets that the user can employ to select and highlight features within the dataset. A relevant aspect of this method is the concept of dual-domain, which connects the spatial and the transfer function domains. However, even for this interactive system proposed by Kniss et al. [65], a considerable expertise from the user is necessary to achieve a meaningful visualization in a reasonable amount of time. It must be noted at this point that, unlike the original approach by Levoy, where this 2D information (intensity value and gradient magnitude) is employed to determine a proper opacity value (1D), in this case the transfer function is a mapping from 2D to 4D,

$$f : \mathbb{R}^2 \mapsto \mathbb{R}^4$$

as the three color components (e.g., RGB) are also to be generated by the classification mapping. This not only increases the degrees of freedom, and therefore the complexity of the function itself, but also represents a challenge in terms of usability, as the design of an intuitive user interface for the definition of such a mapping is by no means trivial. In a medical scenario, this would imply the necessity of a computer graphics expert assisting the physician during the transfer function definition, which is not practicable in most situations.

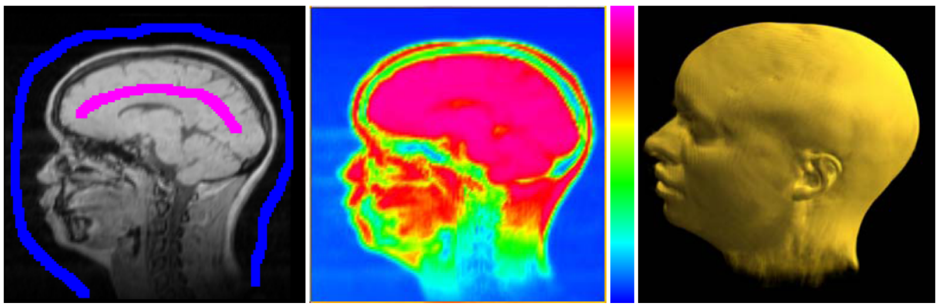


Figure 4.12: Example of transfer function definition based on paintings done by the user on slices and neural networks [125].

The concept of direct interaction with the rendered result instead of with a representation of the transfer function domain has been taken one step further by Tzeng et al. [125, 126], as shown in Figure 4.12. In this case, the transfer function space is kept completely hidden to the user, who only interacts with the volume itself by painting on sample slices of the dataset. The classification itself is performed by one multilayer perceptron (MLP) neural network for each predefined material class.

Partially inspired by this work, we have developed our own interface for volume classification specially oriented to medical visualization [8, 7]. In our solution, we borrow the idea of using a multi-dimensional transfer function, while limiting the user interaction to the spatial domain. We also combine the result of the user interaction with machine learning methods in order to produce the final transfer function in an automated way. However, our approach clearly differentiates from the one by Tzeng et al. [125, 126] by employing an augmented reality paradigm on which a real 3D interaction with the volume is guaranteed, in contrast with a 2D slice-based solution. Moreover, our automatic classification process defines a standard multi-dimensional transfer

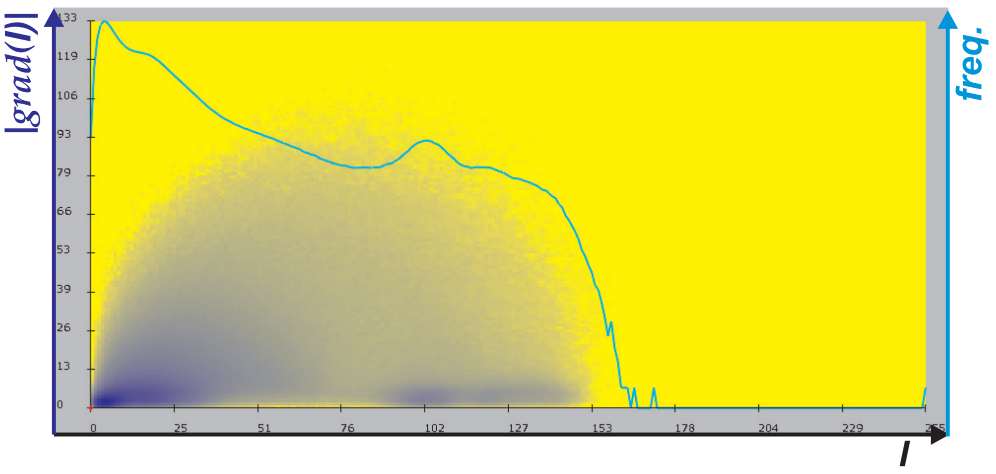


Figure 4.13: Example of 2D histogram. The cyan line shows the standard (accumulative) 1D histogram on logarithmic scale. Each of the points forming a cloud represents one voxel in the volume positioned according to its intensity (X coordinate) and gradient magnitude (Y coordinate).

function for the whole volume and does not require an extra neural network for each sought after material.

In our strategy, a 2D transfer function is generated in a semiautomatic way. The transfer function itself can be represented as a mapping from a voxel's intensity, I, and gradient magnitude, ∇I, to the opacity, α, and color (r, g, b), that are to be utilized for rendering that given voxel. It is, therefore, a mapping from 2D I,∇I to 4D r, g, b, α. As mentioned above, the representation of such a function is not trivial. In our case, in order to present and analyze the results, the generated transfer function is represented on top of the 2D histogram of the dataset. A mixed representation of a 2D histogram together with a regular 1D histogram is shown in Figure 4.13. As can be seen in the figure, the X-axis represents the voxel intensity, while two Y-axes are utilized. On the right hand side, the frequency with which each intensity value appears in the volume (on logarithmic scale) is represented, so that a cyan line depicts the traditional 1D histogram. The left Y-axis is used for the representation of the gradient magnitude in order to generate the 2D histogram. In this case, each voxel is depicted as a 2D point using its $(I,\nabla I)$ as coordinate values, hence producing the cloud of points that can be seen in the figure. This way, the frequency of appearance for each pair $(I, |\nabla I|)$ is indicated by the density of this cloud at each position on the 2D histogram. In the remainder of this chapter, all 2D histograms are presented using this abstraction, representing in every case the 1D histogram with a cyan line and the 2D histogram with a cloud of points. Further details about this type of representation are provided in Section 4.2.1.3, when the classification results are presented using 2D histograms as a basis.

4.2.1 AR-Based Semiautomatic Transfer Function Definition

As briefly pointed out above, in direct volume rendering in general, and in medical volume visualization in particular, a proper transfer function is vital for producing a meaningful image that provides a good insight of the data. Since in most cases the information sought after is highly dependent on the application, user interaction becomes a crucial factor to allow the user to guide the classification of the dataset. Therefore, our novel approach translates the interaction between the user and the analyzed data into an augmented reality environment, so that a better and more direct manipulation of the volume is enabled.

The term *augmented reality* (AR) denotes techniques which combine images of the real environment with three-dimensional computer-generated graphics. A thorough survey of augmented reality falls beyond the scope of this thesis, but an overview is given by Azuma [23].

Augmented reality user interfaces have recently been used as a tool for defining transfer functions in a manual way [104]. However, it must be noted that the proposed paradigm corresponds to a traditional one-dimensional transfer function, where the user manually combines a set of predefined functions in a trial and error manner. This way, an opacity transfer function can be determined for a gray-scale representation. This clearly differentiates from our volume classification approach, which focuses on the combination of multi-dimensional transfer functions with a user friendly AR-based user interface. This type of interaction has the advantage of providing a better understanding of the actual three-dimensional structure of the dataset, enabling a deeper integration of the user into the transfer function specification process. The actual transfer function design is kept transparent to the user by employing machine learning classifiers, thus freeing the user from the internal complexity of the classification.

Our semi-automatic volume classification proposal has been implemented using our own medical augmented reality framework, called ARGUS [47, 48]. Following, relevant aspects about medical augmented reality and the ARGUS system are summarized.

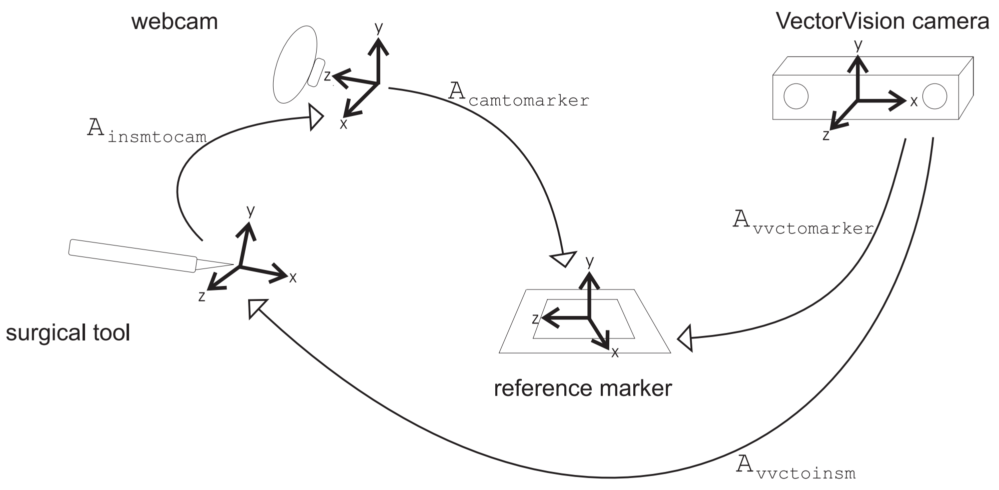
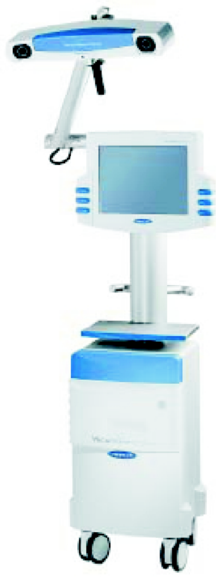


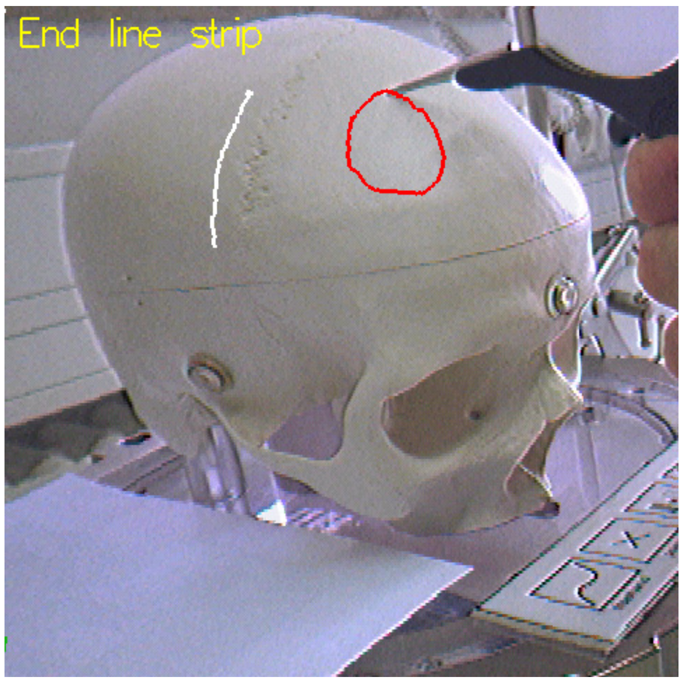
Figure 4.14: Overview of the coordinates transformation involved in the tracking process [47].

4.2.1.1 Medical AR

The application of augmented reality in medical diagnostics and treatment has been in the focus of active research for many years. Among the most relevant references is the work by State et al. [118], who presented an early system for supporting ultrasound-guided needle biopsies. Figl et al. described a head-mounted operating microscope which is capable of overlaying additional graphical information over the conventional microscopic view [45]. More recently, a high performance video see-through augmented reality system for medical applications was presented by Vogt et al. [128]. Also well known is the *Studierstube* project, conceived and developed by Schmalstieg et al. as a collaborative augmented reality system [111].

Our transfer function design process uses an augmented reality environment for displaying the volume datasets. Moreover, intuitive 3D user interaction is provided by a specialized AR-based user interface. The system has been realized using our framework for medical augmented reality, ARGUS (***A***ugmented ***R***eality based on Image ***GU***ided ***S***urgery). Unlike many other experimental setups for medical augmented reality, ARGUS uses existing, commercially available medical equipment [47]. In the AR setup, a VectorVision® intraoperative navigation device is employed (see Figure 4.15(a)), which is equipped with a highly accurate infrared tracking system. The tracking information delivered by the infrared cameras is utilized for the pose estimation of the digital video camera used in the AR system, and for realizing the novel user interaction in our semi-automatic vol-


(a) VectorVision®



(b) Example application.

Figure 4.15: Medical AR. (a) IGS device produced by the *BrainLAB* company (Heimstetten, Germany). (b) 3D user interaction based on intraoperative navigation: operation planning drawings on a plastic skull phantom [46].

ume classification approach. An illustration of the transformation matrices involved in the tracking and calibration process is presented in Figure 4.14. Further details about the medical AR set-up can be found in the work by Fischer et al. [47].

Several objects can be tracked simultaneously by the intraoperative navigation system. Infrared marker clamps consisting of a configuration of three reflective spheres are attached to surgical or interaction tools which are to be tracked. Moreover, a pre-configured pointer tool is supplied by the manufacturer of the IGS device. Using these capabilities, a user interaction library based on this medical AR framework has been designed and implemented [46]. The library allows the replacement of traditional input devices such as keyboard or mouse with direct tools in a medical AR application. We use different pen-like and pointer-like tools as wireless interaction devices. The user interface system is capable of detecting different click gestures for the definition of points in 3D. Moreover, a standard menu system with freely placeable menu items is provided. An example application of the interaction

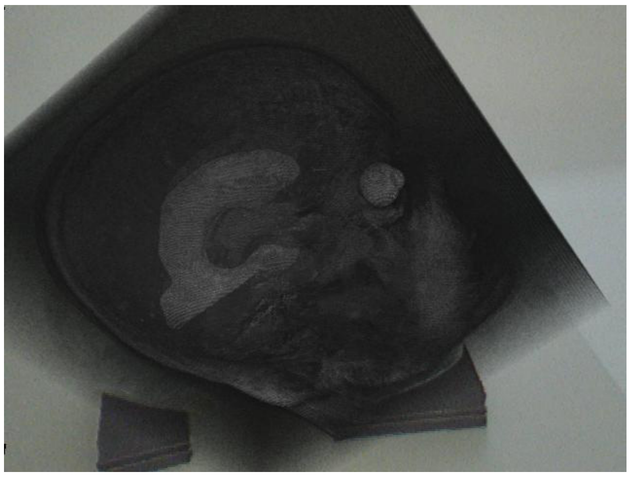


Figure 4.16: Volume rendering of the patient's dataset in an AR environment prior to the application of the volume classification algorithm.

library is illustrated in Figure 4.15(b). Further information about the user interaction library and examples can be found in [46].

4.2.1.2 Volume Classification

The novel paradigm for volume classification we propose constructs a multi-dimensional transfer function in a semi-automatic manner. The key elements of the algorithm are summarized in a functional pipeline:

- Render volume in AR environment: The original dataset is rendered with a standard linear ramp transfer function for all color and opacity channels producing a gray-scale representation. The obtained result is displayed using 3D texture mapping (see Figure 4.16).
- Inspect volume: The rendered volume can be directly examined in the AR environment using a clipping plane widget (see below).
- Select sample points: Combining the use of the clipping plane widget together with a pointer tool, sample points representing features of interest can be easily selected.
- Generate transfer function: The information corresponding to the selected points is processed and utilized to automatically generate an appropriate transfer function. This automatic process is achieved with

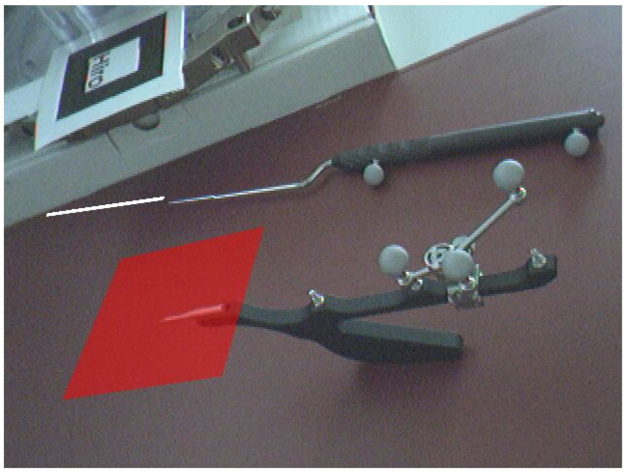


Figure 4.17: AR interaction tools: clipping plane (lower image) and pointer tool (upper image) [8].

the help of machine learning algorithms on the basis of a 2D histogram (voxel intensities and gradient magnitudes) (see Section 4.2.1.2.2).

- Render classified volume: Once a satisfactory transfer function has been obtained, the volume is classified and a final rendering image is produced.

Following, a detailed description of the two most relevant elements of this pipeline is presented.

4.2.1.2.1 Sample Points Selection In our system, the user can directly manipulate and interact with the volume in an AR environment. The correct positioning of the scanned dataset (CT, MRI) is guaranteed by the registration process provided by the image-guided surgery (IGS) system [47]. Two intuitive tools are then employed by the user to inspect the volume and define regions of interest. Figure 4.17 shows both interaction tools together with their virtual representation immersed in an AR environment: a clipping plane and a pointer tool.

The clipping plane tool (see the tool and the virtual plane in red on the lower part of Figure 4.17) is tracked with the help of three non-aligned reflecting spheres (6-DOF) in such a way that its position and orientation can be retrieved from the IGS system. With this information, a plane containing

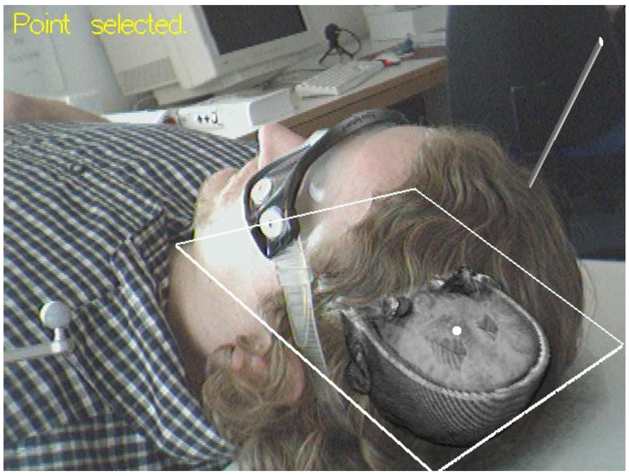
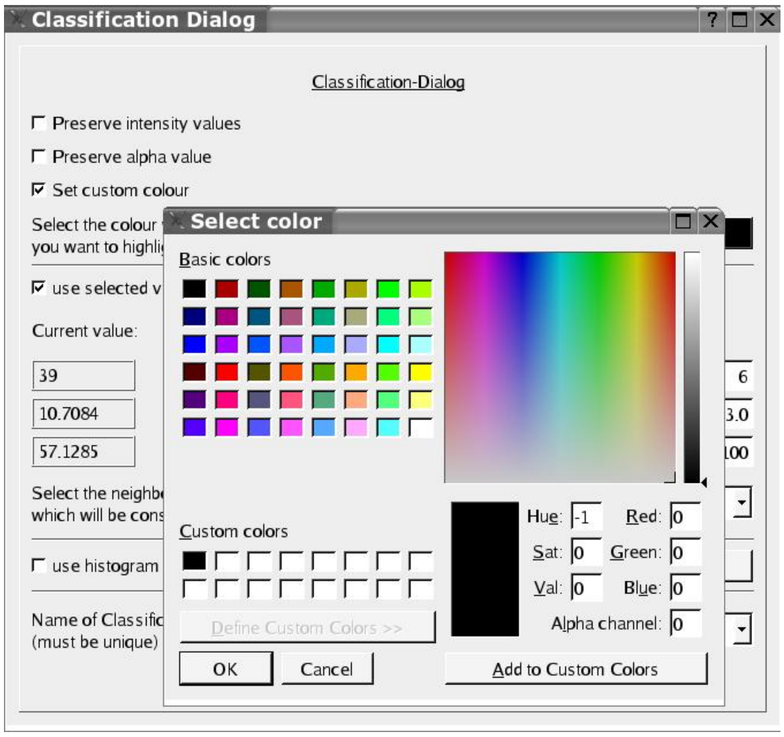


Figure 4.18: Sample point selection [8].

the three reflecting spheres can be easily modelled. An intersection test between this plane and the volume bounding box provides the information necessary to cull off all voxels *above* the defined plane. This way the user can browse through the volume rendered in the AR environment by just moving the clipping plane tool in any direction, providing a direct insight of the data.

The second interaction tool is a pointer. As can be seen in the upper part of Figure 4.17, the pointer tool is tracked using two spheres (5-DOF). The position of these two points determine a vector that can be used to represent the direction of a straight line (the virtual extension of the pointer is shown in white in the figure). A simple plane-line intersection test between this straight line and the plane defined by the clipping plane tool can be used to indicate a position within the dataset. In order to complete the type of interaction needed by our system, we need to be able to generate an activation *click* event informing that the current position corresponds to a region of interest and should be used for the transfer function design process. We solved this problem with our AR user interaction library (see Section 4.2.1.1). By holding the pointer with a fixed orientation during a predefined time (1 ~2 seconds), the user generates an event informing the system that the point selection process must be activated.

Once a point has been selected (see Figure 4.18), a dialog allows the user to select color and opacity values (see Figure 4.19), as well as an identifying name for the class corresponding to the current material. These material attributes (color, opacity and name) are saved for further use, and a list with all previously defined materials is generated and presented to the user.



☐ Preserve intensity values
☐ Preserve alpha value
☑ Set custom colour
Select the colour
you want to highli
☑ use selected v
Current value:
39
10.7084
57.1285
Select the neighb
which will be cons
☐ use histogram
Name of Classific
(must be unique)

Hue: -1 Red: 0
Sat: 0 Green: 0
Val: 0 Blue: 0
Alpha channel: 0

Figure 4.19: Color selection for chosen material [8].

This way, the point selection can be repeated either to add new points to an already defined material class (by choosing the material name from the list), or to introduce new materials that should be taken into account during the classification of the volume (by entering a new name on the list and defining the corresponding color and opacity material attributes).

4.2.1.2.2 Transfer Function Generation The actual transfer function generation process utilizes the information provided by the user, i.e., the set of sample points and the material attributes associated to them, for the classification of the volume. The basic idea of this semiautomatic approach is to free the user from the more cumbersome aspects of the classification process, while still letting him/her control the final visualization result. So, intuitive parameters such as the material attributes (color and opacity) are directly entered by the user and linked to regions in the volume during the sample points selection. On the other hand, the actual mapping of intensity and gradient magnitude intervals to renderable attributes (i.e., the material attributes defined by the user) is carried out in an automatic manner, due to the difficulties of designing such a transfer function. By adopting this type of approach for the transfer function definition, the initial complexity of the function itself is reduced from a general $f : \mathbb{R}^2 \mapsto \mathbb{R}^4$ to a more simple mapping for which the original range has been restricted to a finite (and usually small) set of 4-tuples, $M \subset \mathbb{R}^4$, where each element $m \in M$ corresponds to a user defined material and contains its color and opacity values.

As mentioned above, the points selected by the user constitute the input to the automatic transfer function generation. In order to minimize the number of points that must be defined by the user and to make the point selection more robust against noise in the dataset and inaccuracies during the point selection, a larger set of sample points is generated out of the initial subset defined by the user. For each point selected with the interaction tools, a small surrounding region around the point is determined. The user has the opportunity to choose between two different specifications for this surrounding region: a 3D neighborhood and a planar neighborhood. The 3D neighborhood is formed by the $3 \times 3 \times 3$ first order neighbors along the three orthogonal directions (X,Y,Z). The planar neighborhood is a two-dimensional 3×3 set of points on the intersection polygon determined by the clipping plane tool. In either case, the computed set of points is taken and the average values for both intensity and gradient magnitude are calculated. This way, even if noise is present in the dataset, the user can rely on the fact that the selected point does correspond to a representative value of the spatial region which has been pointed to.

Next, for each of these average points representing the region selected by the user, a small two-dimensional interval in the transfer function domain is taken around the given values, and the cartesian product of these two subsets is employed to generate the set of points that are passed to the actual classifier.

$$[I_i - \Delta I, |\nabla I_i| - \Delta|\nabla I|] \times [I_i + \Delta I, |\nabla I_i| + \Delta|\nabla I|] \tag{4.4}$$

Equation 4.4 indicates this set of points, where I_i and $|\nabla I_i|$ denote, respectively, the voxel intensity and the gradient magnitude of the average point representing one region selected by the user.

After multiple tests, an interval width ($2\Delta I$) of 12 HU (Hounsfield units) in intensity and 6 in terms of gradient magnitude ($2\Delta|\nabla I|$) were found to be a suitable value in order to obtain a proper representation of the selected points. Hence, 72 sample points are generated for each user selection, ensuring a sufficient amount of input data for the automatic classification process. Due to this procedure, the spatially guided selection performed by the user is combined with a data centric generation of similar points in the transfer function domain.

Machine Learning Classifiers The transfer function generation is performed automatically using machine learning methods. *Machine Learning is the study of computer algorithms that improve automatically through experience* [91]. In our case, this experience is provided by the set of points and materials defined by the user, which establish the starting point for the automatic transfer function generation. Since the initial correspondences between these sample points and the material attributes they are to be mapped to are already known, the actual transfer function design can be reduced to a partitioning of the 2D histogram of the dataset among regions corresponding to the different materials defined by the user. Therefore, automatic classification and spatial partition methods may be used here for the determination of an appropriate transfer function. Two examples of such approaches have been tested in this work. The first is based on the utilization of an artificial neural network, more specifically a *Multi-Layer Perceptron (MLP)*, while the second employs a *k-Nearest Neighbors (kNN)* classifier.

Multi-Layer Perceptron Classifier Artificial neural networks are composed by simple processing elements (artificial neural cells) organized in architectures characterized by a high degree of interconnection inspired by the parallel architecture of animal brains. They result specially interesting for classification purposes due to their ability to approximate functions based on sparse data through a training process and to apply this to solve new problems of similar nature. This training process adapts the weights modulating the value across each connection until the network implements a desired function.

For our purposes, we have chosen a three-layer perceptron topology, using the supervised training method known as *Feed-Forward Back-Propagation* algorithm. Figure 4.20 shows an illustration of such a network. Our network is composed by one input layer of two cells, one for each value in the domain of the transfer function (I_i, $|\nabla I_i|$), one output layer with as many cells as material classes have been defined by the user, and one hidden layer. After several tests, we have determined that a hidden layer of 15 cells is able to properly discern among the user defined classes (typically $4 - 8$ materials).

For the implementation of the neural network, a third party library (LTI-Lib from the University of Aachen (Germany)) was selected. The functioning principle is simple: The set of sample points created out of the user selected

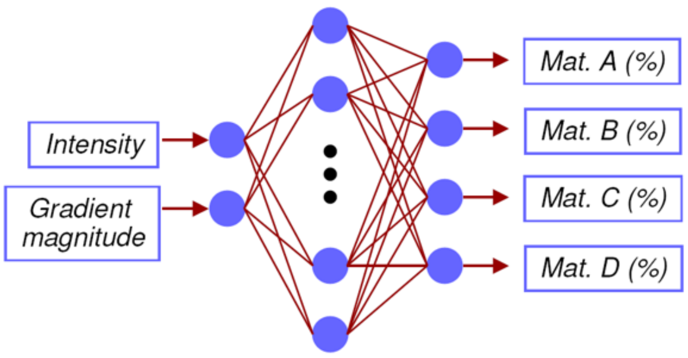


Figure 4.20: Structure of a three-layer perceptron [8].

positions is utilized as a training set for the neural network. Once the network has been properly trained, each voxel intensity and gradient magnitude are fed to the network and an output value is produced at each cell of the output layer. As mentioned above, the output layer has one cell for each material class. The value generated at each of these cells indicates the probability that the input voxel belongs to the material associated to the output cell. This way, a threshold can be set so that each voxel producing a probability over the threshold be classified with the color and opacity of the corresponding material, while the remaining voxels with probabilities below the threshold can be either ignored ($opacity = 0$), or classified with the initial standard transfer function.

k-Nearest Neighbors Classifier A k-nearest neighbors (kNN) analysis provides a different alternative for the automatic classification of new objects out of a number of known examples. In our approach we use a 1-nearest neighbor classifier ($k = 1$). The operating routine is very simple, though effective. The known examples constitute classes defined by the position of a *prototype*. The initial position of each prototype is given by the first sample point corresponding to a material class being passed to the input of the classifier. Each subsequent sample point is assigned to the class whose prototype is at the lowest Euclidean distance computed on the transfer function domain. Once the assignment has been completed, the prototype is recomputed as the average position of all the points belonging to the class it represents. Alternatively, each new sample point may be considered as a prototype of a new class, resulting in a cluster of sibling regions for each material. In either case, with every new sample point, the same process is repeated until all the points generated as a result of the direct user interaction have been classified, and the corresponding prototypes have been respectively created and repositioned to their final location. Then, the kNN (i.e., 1NN) classifier can be utilized to perform the classification of the volume, voxel by voxel.

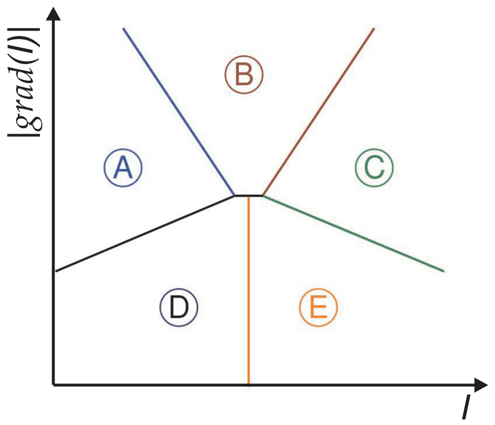


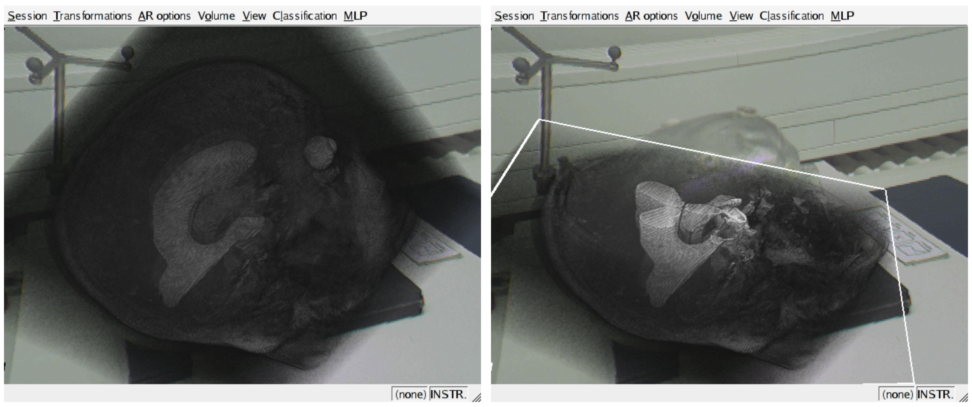
Figure 4.21: Structure of a kNN classifier (k=1). The red point illustrates an example of a voxel belonging to class B [8].

Figure 4.21 illustrates a simple example of this process. The labels A, B, C, D, E show the prototype positions corresponding to five different material classes selected by the user with the AR-based interaction tools. In this scenario, each voxel is classified as belonging to the material class whose prototype is the closest in terms of intensity and gradient magnitude. The red point in Figure 4.21 represents a voxel being classified. Since its nearest neighbor ($k = 1$) is prototype B, the voxel will be classified with the color and opacity associated to this material.

Even though the kNN classifier does not provide a direct measure of the probability for each voxel to belong to a specified material class, the distance to the closest prototype can be used for this purpose. This way, it is possible to set a threshold again in order to decide whether all voxels should be classified (assigned to one of the defined classes), or those which are not close enough to any prototype should be specially handled (e.g., ignored — $opacity = 0$ — or classified with the initial gray-scale linear ramp transfer function).

4.2.1.3 Results

A prototype implementation of our proposed method has been realized in a master project (*Diplomarbeit*) [67]. Based on this prototypical implementation, the viability of the proposed method has been tested on several scanned medical datasets (CT, MRI). Here, two representative examples are presented in detail. Other results are then briefly summarized with illustrative images.


Session Transformations AR options Volume View Classification MLP
(none) INSTR.
Session Transformations AR options Volume View Classification MLP
(none) INSTR.

(a) External view. (b) Insight view.

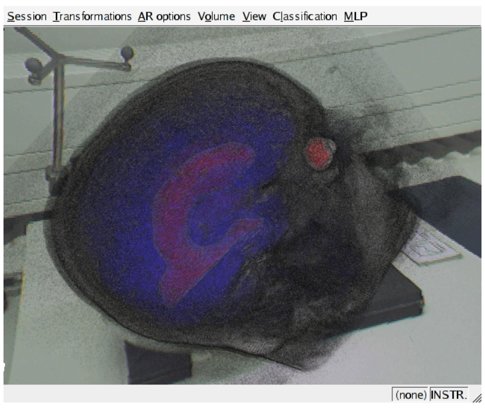
Figure 4.22: Unclassified MRI head overlaid on top of plastic skull phantom.

The first example dataset is an MRI scan of a human head with 512 × 512 × 160 voxels and 8 bits/voxel of resolution. In order to simulate the typical scenario for a medical AR application, the patient's scan has been registered using a plastic skull phantom. This way, the patient data can be efficiently overlaid on top of the phantom as presented in Figure 4.22.

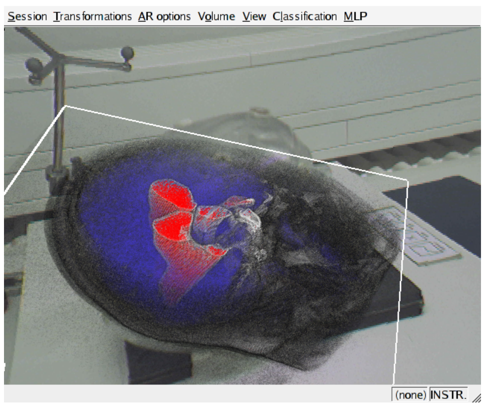
Material	Color	Opacity	# Points
air	black	0.0	5
fat	yellow	10.0	6
inner tissue	blue	20.0	3
ventricle	red	120.0	5
edges	white	20.0	6

Table 4.1: Material classes defined for the MRI head dataset shown in Figure 4.22.

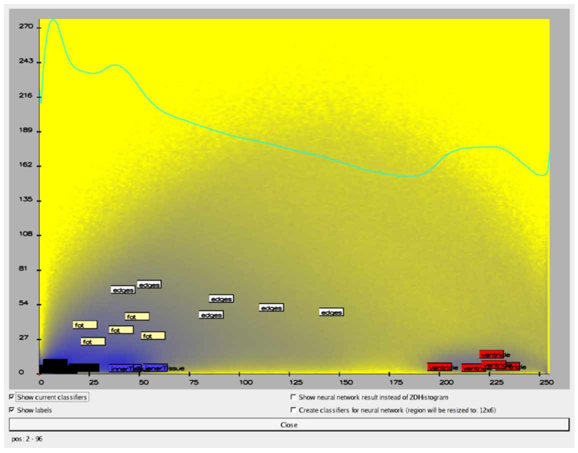
Figure 4.22 shows the starting point for our approach. The volume is first rendered in an AR environment using a standard linear ramp one-dimensional transfer function for all color and opacity (alpha) channels, where only the voxel intensity is taken into account. The volume rendering is performed with 3D texture mapping using pre-classification to apply the transfer functions. As can be seen on the images (see Figures 4.22(a) and 4.22(b)), the initial rendering does not provide a sufficient insight of the structures present in the


Session Transformations AR options Volume View Classification MLP
(none) INSTR.

(a) External view.


Session Transformations AR options Volume View Classification MLP
(none) INSTR.

(b) Insight view.



(c) 2D histogram with user defined sample points.

Figure 4.23: Intermediate classification with user defined sample points on top of plastic skull phantom.

MRI scan of the patient's head, and a new classification must be performed in order to reveal the information it contains. Using the direct interaction tools (clipping plane and pointer tool), we define five different materials as described in Table 4.1[4], which indicates the user defined material attributes and the number of points, also selected by the user, for each of these materials.

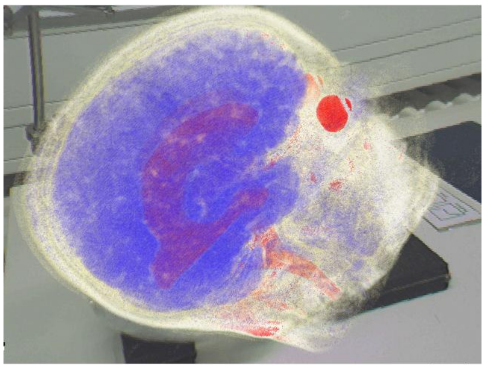
Each material is represented by a color and opacity value, as well as an identifying label. The selected sample points are reproduced in Figure 4.23(c)

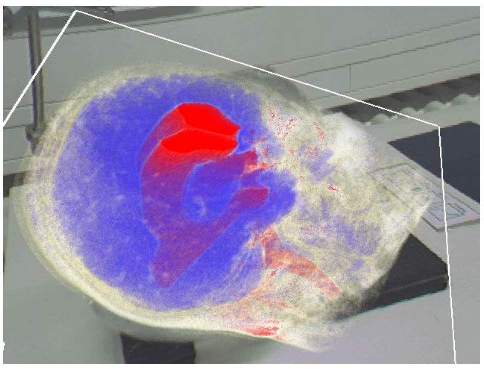
[4]The opacity values in this table are unclamped alpha values which are then automatically clamped to the range $[0, 1]$ by OpenGL.

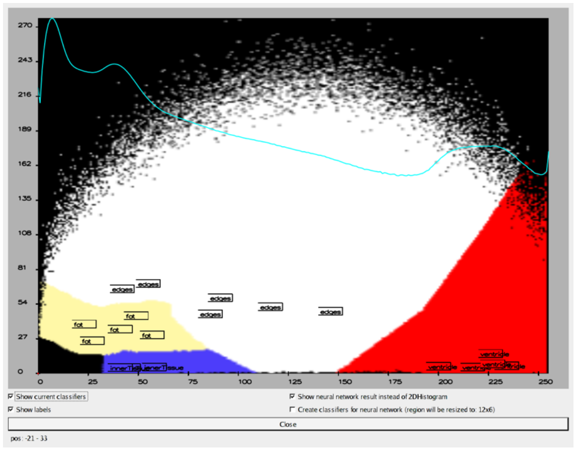
on the 2D histogram of the dataset. Each colored rectangle illustrates the size of the intervals in terms of intensity and gradient magnitude employed for the generation of the input to the automatic classifiers. Since these labelled regions define a color and opacity for those voxels represented below the respective areas on the histogram, a preliminary manual classification may already be performed with these user defined data. The result of such an intermediate classification is shown in Figure 4.23(a) and 4.23(b). Even though this already provides a more appealing visual result than the initial standard transfer function, the result is still not satisfactory enough for most applications. Of course the user could further refine this intermediate manual classification by defining more and more points for each material. This, although compatible with our algorithm, would notably increase the time required for the classification of the volume, thus limiting the benefits introduced by the method. Therefore, an automatic classifier is used instead. This way, a minor set of points is enough to create a much more complex and meaningful visualization of the patient's anatomy.

In this example, we utilize both automatic classifiers (MLP and kNN), so that a comparison of the pros and cons associated with them can be performed. First, the k-nearest neighbors classifier is applied. The setup in this case consists in the creation and proper allocation of the class prototypes. Given the reduced amount of sample points (25) that must be processed and the limited number of material classes (5), this initialization step is accomplished almost instantaneously. For this example no threshold was set, letting the classifier act over the whole volume. This way, each voxel is designated to belong to one of the user defined materials. The obtained result is illustrated in Figure 4.24. Figures 4.24(a) and 4.24(b) clearly show the ability of the kNN classifier to emphasize those features explicitly selected by the user in the AR environment. Furthermore, the dark cloud produced by the air surrounding the head has also been effectively removed. The transfer function itself is represented in Figure 4.24(c) on the 2D histogram of the dataset. Each point on this 2D histogram corresponds to one voxel in the volume, while the colors indicate the distribution of the classes (i.e. materials) produced by the automatic classification process. This image clearly illustrates how unintuitive the interaction on a histogram basis would be, compared to the direct selection of points in the volume.

In order to confront the results produced by the kNN classifier with those of a neural network, the same set of points selected for the first case (see Figure 4.23(c)) was employed to train the MLP neural network. The training set for the neural network is generated as described above. For this example, we have configured the training process to a maximum length of 400 epochs


(a) External view.

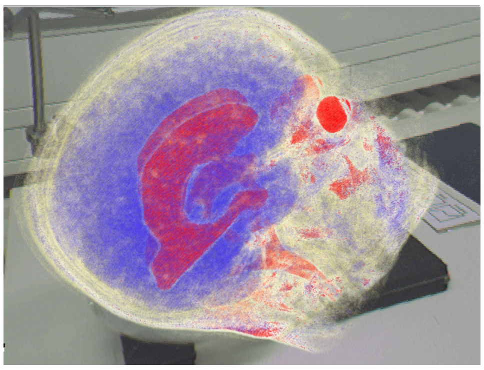

(b) Insight view.



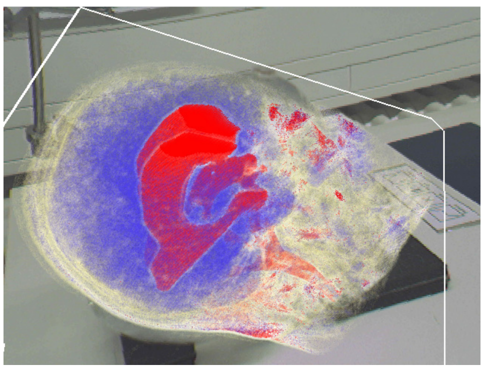
☑ Show current classifiers
☑ Show labels
☑ Show neural network result instead of 2DHistogram
☐ Create classifiers for neural network (region will be resized to: 12x6)
pos: -21 - 33

(c) 2D histogram showing transfer function.

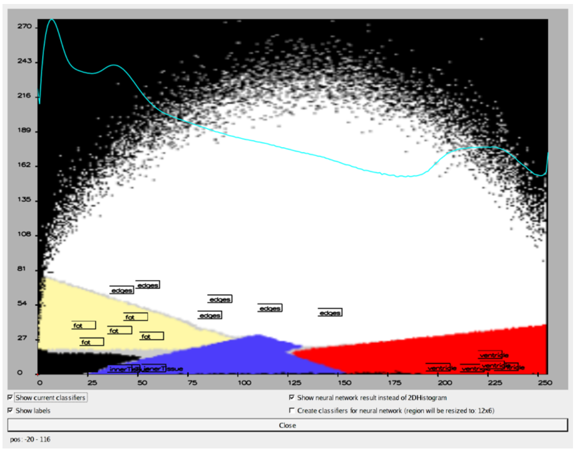
Figure 4.24: Result of automatic classification with kNN classifier on top of plastic skull phantom. Each material is rendered with the color and opacity defined by the user. The 2D histogram shows the generated transfer function where each color identifies one of the given materials.



(a) External view.



(b) Insight view.



[✓] Show current classifiers
[✓] Show labels
[✓] Show neural network result instead of 2DHistogram
[ ] Create classifiers for neural network (region will be resized to: 12x6)
pos: -20 - 116

(c) 2D histogram showing transfer function.

Figure 4.25: Result of automatic classification with MLP neural network on top of plastic skull phantom.

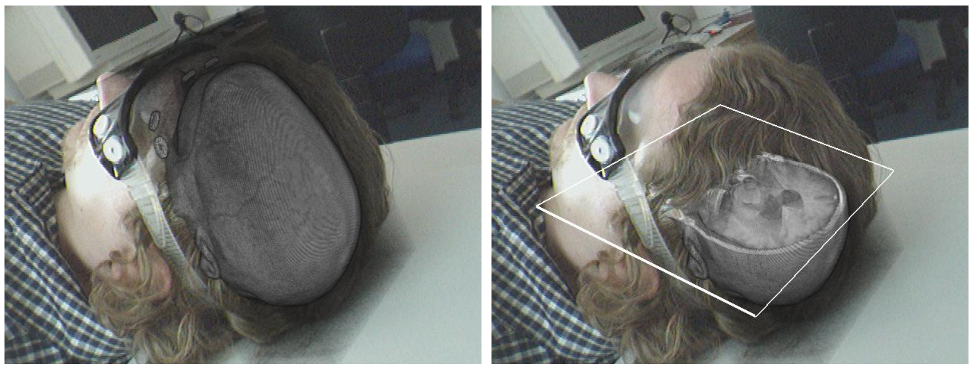
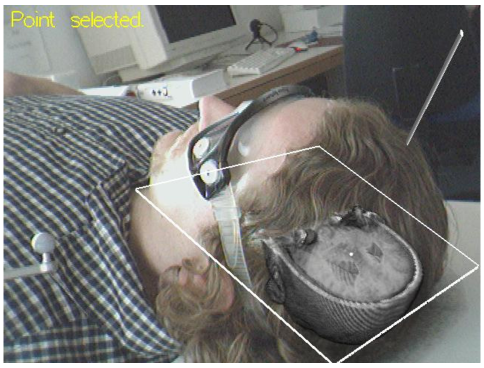

(a) External view. (b) Insight view.

Figure 4.26: Unclassified MRI head overlaid on top of test subject's anatomy.

(iterations of the back-propagation algorithm). Under these conditions, the network was trained after approximately 2 minutes[5]. In this case, a threshold of 70% was set as the minimum probability to consider a voxel as belonging to one of the defined material classes. Voxels with a probability inferior to 70% were rendered as transparent. Figures 4.25(a) and 4.25(b) show the patient's scan rendered after being classified by the MLP neural network. The generated transfer function is also depicted in Figure 4.25(c) indicating all the different colored regions on the 2D histogram of the volume.

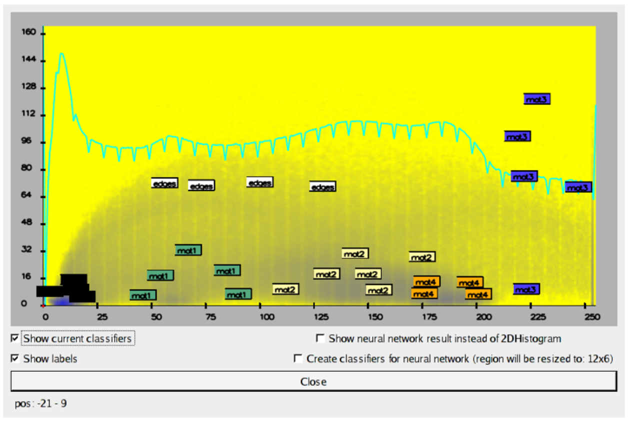
Comparing the results obtained with both classification methods (see Figures 4.24 and 4.25), the outcome produced is surprisingly similar for both, the MLP artificial neural network and the kNN classifier, specially considering their completely different internal structure. Both approaches have been proven to be effective removing undesired elements of the dataset (e.g. surrounding air) and also highlighting materials and features of interest (e.g. ventricular system) over uninteresting background regions. However it is also possible to recognize some characteristic differences between them. The artificial neural network, probably due to its more complex structure, produces a slightly more accurate classification of the data than the k-nearest neighbors classifier. This is particularly true for fine structures and material interfaces like the contour of the ventricle system. On the other hand, the simplicity of the kNN classifier makes it also attractive given its shorter runtime and similar visual performance. In any case, further testing should be performed in collaboration with our clinical partners before a definitive evaluation of the utility of both classifiers can be proclaimed, as at the current state both

[5]Our test system is a PC with an Intel® Xeon™ processor running at 2.66 GHz and a graphics card based on an NVidia® GeForce™FX 6800 chipset.



(a) Point selection.

(b) External view of interaction during point selection.



☑ Show current classifiers
☐ Show neural network result instead of 2DHistogram
☑ Show labels
☐ Create classifiers for neural network (region will be resized to: 12x6)
Close
pos: -21 - 9

(c) 2D histogram showing selected sample points.

Figure 4.27: Selection of sample points.

alternatives have proven their adequateness. It can be stated, though, that the divergence observed between both classifying methods tends to diminish with the amount of selected sample points. This is the expected course, since a higher number of sample points implies more control supplied by the user during the transfer function design process, thus constraining the behavior of the automatic classifiers.

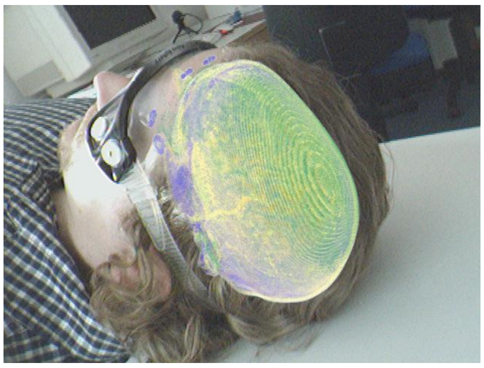
For the second example, we have reconstructed a more realistic scenario, reproducing a situation much closer to the actual application of our proposed method for clinical purposes. Here again, the dataset is an MRI scan of a human head with a spatial resolution of $512 \times 512 \times 74$ voxels with 16 bits/voxel. The MRI scan is first rendered on top of the image of the registered test subject, with help of the infrared tracking provided by the IGS system (see Figures 4.26(a) and 4.26(b)). A set of points is then directly defined in the medical AR environment as indicated in Figure 4.27. Figure 4.27(b) shows an external view of the experimental setup, illustrating the direct interaction process during the selection of a sample point. All the user selected sample points can be seen in Figure 4.27(c) on top of the volume's 2D histogram. The corresponding color and opacity parameters are summarized in Table 4.2.

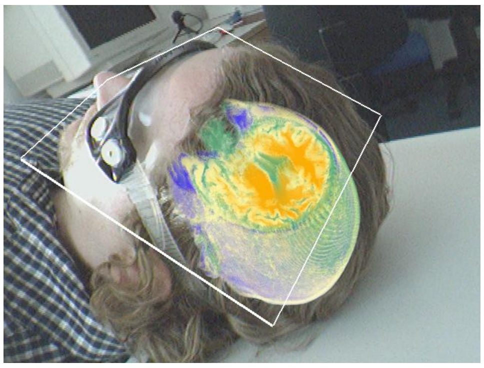
Material	Color	Opacity	# Points
air	black	0.0	4
material 1	green	40.0	5
material 2	yellow	30.0	6
material 3	violet	30.0	5
material 4	orange	30.0	4
edges	white	20.0	4

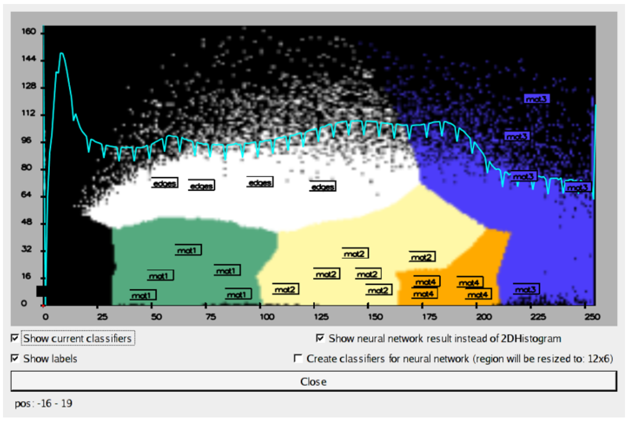
Table 4.2: Material classes defined for the MRI head dataset shown in Figure 4.26.

For this example a kNN classifier was selected for generating the transfer function. As can be seen in Figure 4.28, the obtained classification clearly reveals the inner structure of the dataset, thus confirming that, with a sufficient amount of sample points, a simple kNN classifier is able to produce an appropriate transfer function. This second example not only illustrates the interaction in a real scenario, but also demonstrates how our proposed method allows to produce rather complex volume classifications, that are capable of discerning between different tissues and materials within a patient's anatomy (see Figure 4.28(c)).

Two additional examples of results obtained with our semiautomatic volume classification approach are shown in Figure 4.29 for another MRI scan of a human head and a CT acquisition of a human thorax. The setup utilized in both cases is analogous to that described above, and therefore will not be repeated here in detail. It must be noted that these two examples do not strictly utilize an augmented reality environment, since the real anatomy is not present in the scene. Nevertheless, they still benefit from the direct user interaction within a three-dimensional virtual environment, even though the orientation may turn to be slightly less intuitive than with AR. Both classifications obtained for the MRI head dataset (see Figures 4.29(a) and 4.29(b)) confirm the similarities between the two automatic classifiers, when enough sample points have been defined (19 points for 5 materials in this case). The main differences observed between the image produced by the kNN classifier and the MLP neural network are mostly due to the fact that, in the latter,


(a) External view.


(b) Insight view.

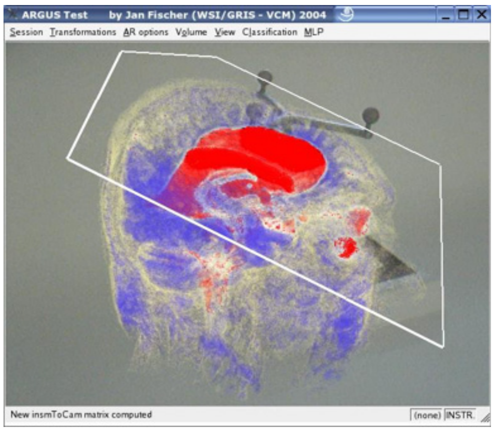


☑ Show current classifiers
☑ Show labels
☑ Show neural network result instead of 2DHistogram
☐ Create classifiers for neural network (region will be resized to: 12x6)
Close
pos: -16 - 19

(c) 2D histogram showing transfer function.

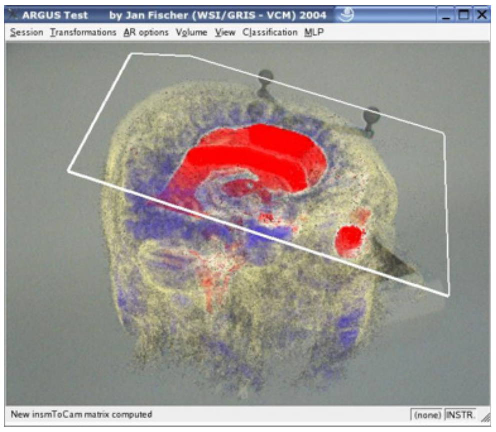
Figure 4.28: Result of automatic classification with kNN classifier on top of test subject's anatomy.

those voxels below the probability threshold were rendered with the initial gray-scale linear ramp transfer function. On the other hand, the CT thorax dataset, given its more complex inner structure, provides a good illustration of the divergences between the classifiers. While the kNN classifier tends to over-classify by assigning every voxel to one of the user defined materials (see Figure 4.29(c)), the MLP neural network is able to better discern fine structures, such as the interface separating bronchi and air in the lungs (see Figure 4.29(d)).


ARGUS Test by Jan Fischer (WSI/GRIS - VCM) 2004
Session Transformations AR options Volume View Classification MLP
(none) INSTR.

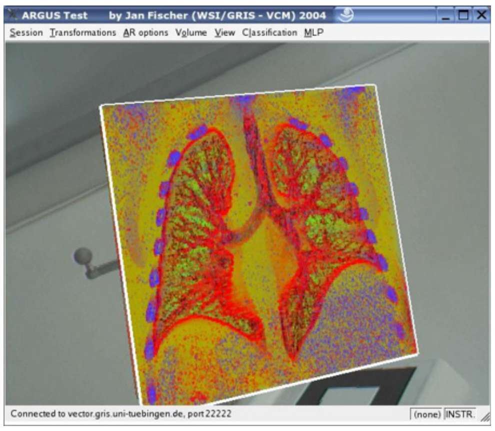
New insmToCam matrix computed

(a) MRI Head (kNN Class.)


ARGUS Test by Jan Fischer (WSI/GRIS - VCM) 2004
Session Transformations AR options Volume View Classification MLP
(none) INSTR.

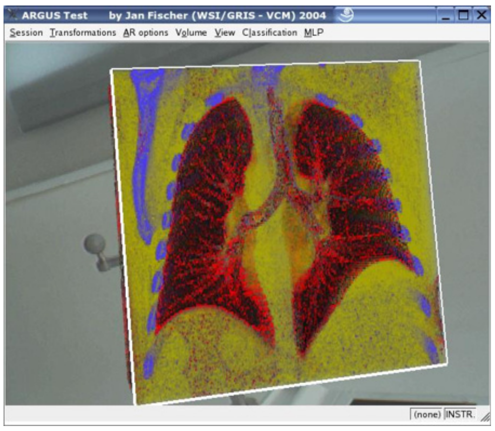
New insmToCam matrix computed

(b) MRI Head (MLP Class.)


ARGUS Test by Jan Fischer (WSI/GRIS - VCM) 2004
Session Transformations AR options Volume View Classification MLP
(none) INSTR.

Connected to vector.gris.uni-tuebingen.de, port 22222

(c) CT Thorax (kNN Class.)


ARGUS Test by Jan Fischer (WSI/GRIS - VCM) 2004
Session Transformations AR options Volume View Classification MLP
(none) INSTR.

(d) CT Thorax (MLP Class.)

Figure 4.29: Results of automatic classification with kNN ((a), (c)) and MLP ((b), (d)) classifiers for an MRI scan of a human head ((a), (b)) and a CT scan of a human thorax ((c), (d)) [8].

4.3 Summary

Direct volume rendering provides a powerful tool for displaying scanned volume data in a three-dimensional environment. Since all the data is directly processed and incorporated to the scene, it generates a valuable global overview of the whole dataset at once. Various are the algorithms to visualize a 3D dataset with direct volume rendering, among which the most important are: ray casting, splatting, shear-warp and texture mapping. In this chapter, the main characteristics of these algorithms have been presented and discussed. Then, once the necessary background has been provided, the focus has been shifted to the problem of classification in volume rendering applications. This consists of the assignment of renderable optical properties to different regions within the dataset. As commonly only certain limited regions of the volume are specially relevant and must be highlighted with respect to the rest of the anatomy, a proper classification becomes a crucial element for a meaningful visualization.

The classification step of the volume rendering pipeline is usually performed by means of a transfer function, which maps internal parameters of the data (e.g., voxel intensity) to color and opacity values to be used during rendering. Much work has been devoted to assist the user in the typically tedious and complex transfer function design. Even though these algorithms have proven their ability to generate appealing images, they require a high level of expertise from the user about the internal classification process. Since this is not the case in most medical scenarios, we have developed an intuitive semi-automatic approach for transfer function definition which is based on direct interaction between the user and the rendered volume in a medical augmented reality (AR) environment. Working on a set of sample points defined by the user directly in the volume, an automatic volume classification process is carried out using machine learning techniques. Specifically, an artificial neural network (multi-layer perceptron) and a k-nearest neighbors classifier have been implemented and tested. Both alternatives have proven to be appropriate for the transfer function design process and their own characteristics have been discussed and analyzed. The results obtained with a prototypical implementation confirm that our novel method permits the generation of complex multi-dimensional transfer functions, producing meaningful images and gaining a better insight of the data. On the other hand, given the combination of an intuitive and easy-to-use user interface and the application of automatic classifiers, the complexity of the volume classification process is kept transparent to the physician, thus facilitating the integration of these techniques in the medical routine as a useful visualization tool.

APPENDIX A

Medical Applications

In this thesis, various innovative algorithms have been presented, which have helped to extend the state-of-the-art of volume visualization in general, and particularly to its application to the medical field. In this appendix, a brief summary of practical projects is exposed, in which volume visualization has played a decisive role. The projects are grouped into two logical blocks: the first focuses on pre- and intra-operative support for medical procedures; while the second includes example applications where isosurface extraction and visualization is the center of attention.

A.1 Visual Assistance for Pre-/Intra-Operative Support

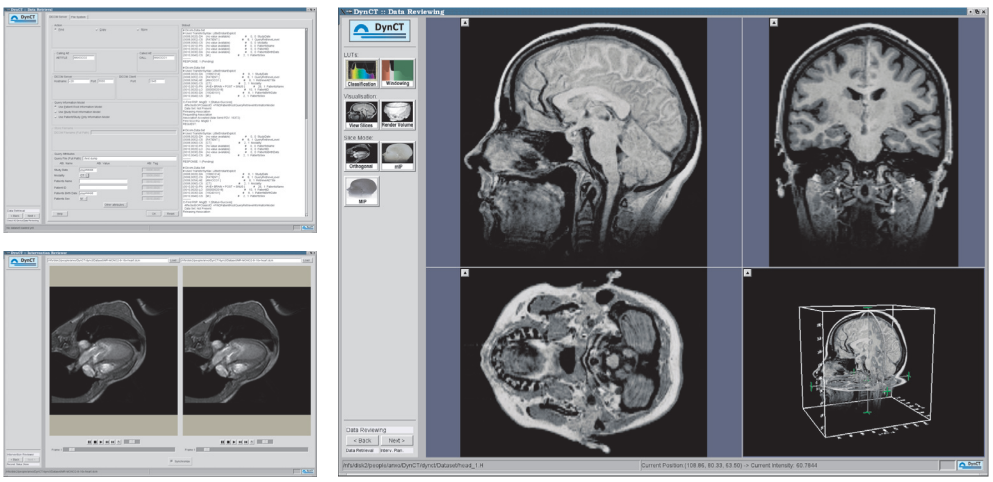
The application of visualization techniques for medical purposes is one of the most active fields for volume visualization. Here, two different projects are presented, which are devoted to support a physician in the preparation and realization of a surgical procedure.

A.1.1 DynCT EU Project: Real Time Motion Compensated Reconstruction and Visualization for Dynamic Computed Tomography

A.1.1.1 Description of the Project

The purpose of the *DynCT* project was to define and develop a novel hardware/-software-based reconstruction and visualization system for real-time medical computed tomography. The system addressed three hot issues in this medical scanning modality: real-time 3D cone-beam reconstruction, organ motion-independent CT reconstruction and real-time 2D/3D visualization.

Within this project, the responsibility of the University of Tübingen was to develop tailored software and hardware for the visualization and interaction with the reconstructed real-time volume data [20].



/nfs/disk2/people/anxo/DynCT/dynct/Dataset/head_1.H

Current Position:(108.86, 80.33, 63.50) -> Current Intensity: 60.7844

Figure A.1: User interface showing the *data retrieval*, *data reviewing* and *reconstruction reviewer* protocol entries.

The main objective for the visualization and user interface development was to deliver high quality images that reveal the structures present in the CT dataset. To make the system applicable for CT fluoroscopy, visualization had to be done in real-time. To handle these requirements we made use of the VIZARDII [89] which is a multipurpose fully programmable PCI card featuring DSP and FPGA components. It has been constructed for volume rendering and delivers high-quality images at a real-time frame rate. The VIZARDII board was originally designed to handle only static volume data. Within the DynCT project, it was required that each frame a different volume dataset be rendered. This made it necessary to extend the original VIZARDII design to handle rendering and uploading of data simultaneously. The implementation of the VIZARDII board is based on a reconfigurable FPGA chip allowing a flexible implementation of different rendering architectures. Therefore the VIZARDII board could be adapted to the input of a new real-time data-interface without the need of redesigning the rendering pipeline.

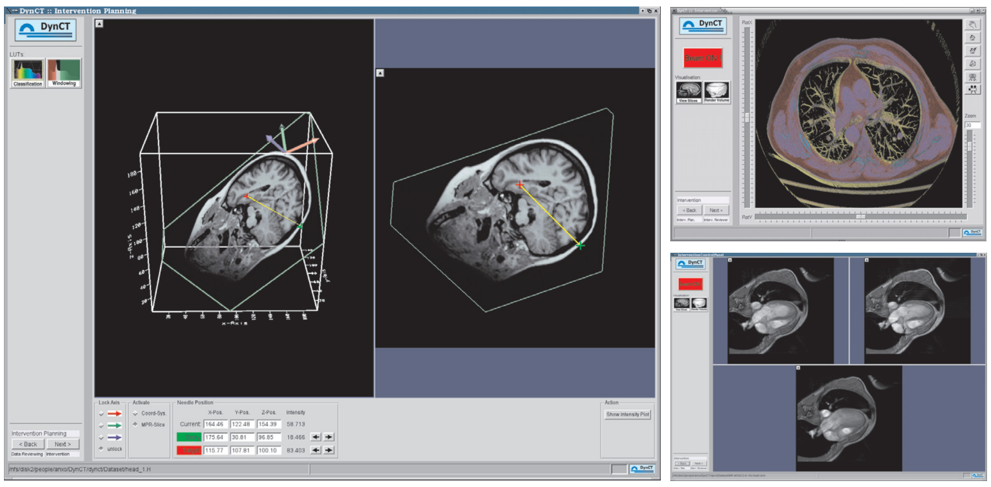


Figure A.2: User interface showing the *intervention planning* and *intervention* protocol entries (with volume rendering and blended slices).

For the user-interaction we developed a framework for an intuitive GUI that controls features such as viewing modality, viewpoints, transfer functions, material properties, cut planes, region of interest selection, reconstruction parameters, intervention planning and intervention guidance (see Figure A.1). The integration of the VIZARDII board as a rendering engine enabled the user to choose between a standard 2D slice view or a 3D volume rendering visualization (see Figure A.2), while using the different protocol entries.

By incorporating new control devices, such as a voice recognition and a remote control system, the user was provided with an extra level of interactivity. This way the physician could handle the user interface while attending the patient.

The DynCT project started in January 2000 and was successfully completed after an extension of 6 months in June 2003.

A.1.1.2 Project Partners

This project was funded by the European Commission as a joint work between the University of Tübingen, Sema Group (Spain), ELTA Electronic Industries Ltd. (Israel), CEA-LETI (France), Philips Medizin Systeme (Germany), Universitair Medic Ctr. Utrecht (Netherlands), Clínica Puerta de Hierro (Spain) and Univ. Politécnica de Cartagena (Spain).

A.1.2 VIRTUE DFG Project: Visualization Platform for Medical Datasets (Helios)

A.1.2.1 Description of the Project

The main goal pursued within the VIRTUE project is the combination of minimally-invasive methods of surgery and techniques from virtual endoscopy and image-guided navigation. The additional information provided by image-guided navigation and virtual endoscopy can improve significantly the three-dimensional spatial orientation during surgical procedures. In particular a better representation of risk structures – like blood vessels or nerve fibers – enables a better planning. Consequently, the risk of serious complications of such an intervention is reduced. An integral part of the project is the development of a software visualization tool, which will be used as a basis for the realization of the navigated virtual endoscopy software.

This software platform, called Helios, initially stemmed from the basis developed in the DynCT project (see Section A.1.1), and has been conceived to fulfill a double purpose. On the one hand, as a framework, it provides common use functionality for volume visualization (i.e. data management), and on the other hand, a viewer has been created to illustrate the use of the framework and to offer the user (physician) a state-of-the-art visualization tool for examining 3D data.

As a viewer, Helios provides a complete set of display modes. The basic mode is classical 3D orthogonal section slices (axial, coronal, sagittal) where the user can interactively browse through the slices along the three main axes. Complementary, several alternative modes have been included, such as maximum intensity projection (MPI) where the maximum intensities along each of the three main directions are projected onto a plane producing X-ray-like images. In Figure A.3, a different example is presented, where the three orthogonal slices are properly placed within a 3D scenario. These can also be combined with an overlaid isosurface to enhance the three dimensional

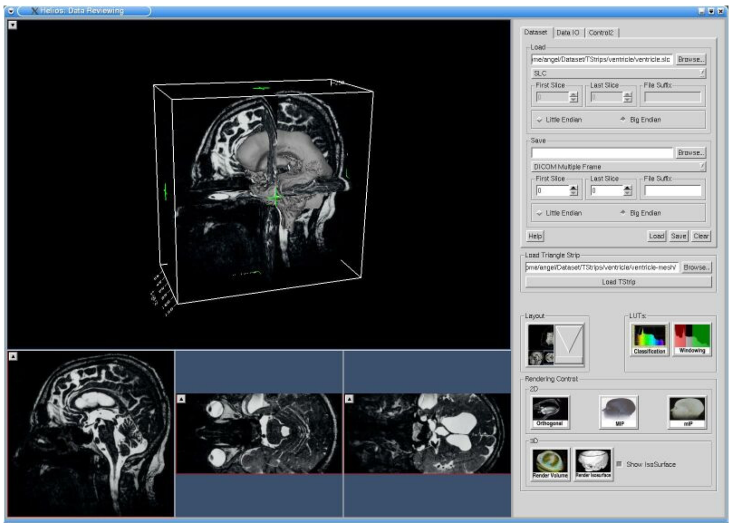


Figure A.3: Helios: User interface showing 3D orthogonal slices and shaded isosurface.

effect (i.e., the ventricle system in the figure).

Based on the experience acquired during the development of a first prototypical version of the Helios viewer, a second version called *Volv*, was implemented in the context of a student thesis [96]. This refactoring effort helped to improve the modularity and flexibility of the software. This is a decisive aspect given the double objective pursued by Helios, as a viewer but also as a framework and software basis. Within this newer version of Helios/Volv, additional display modes have also been added to the viewer. An example can be seen in Figure A.4, where a sequence of slices along the selected direction (axial, coronal or sagittal) are displayed.

Apart from new visualization modes, another important improvement in the new software is the introduction of an enhanced DICOM manager. This new DICOM manager allows a much more intuitive and complete interaction with the datasets acquired by the scanner, by automatically sorting the data according to the meta-information contained in each DICOM-header.

An example of the capabilities of Helios/Volv as a software framework can be seen in Figure A.5. This illustrates a collaboration project with the Department of Maxillofacial Surgery from the University of Tübingen. The objective in this case is to perform a volume transformation so that the final volume is a mirrored version of the original dataset. In order to support such type of functionality within the software framework, a general plug-in concept has been added to the processing pipeline. This allows a

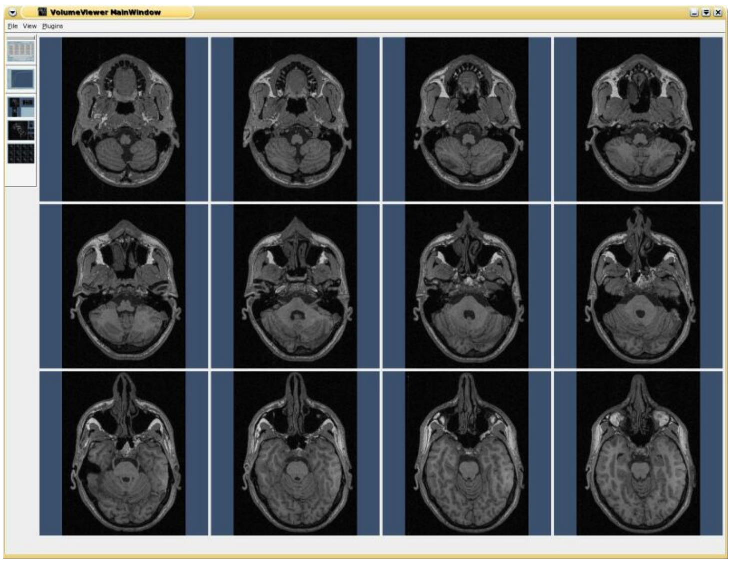


Figure A.4: Volv: User interface showing a sequence of slices along one of the main axis (axial, coronal, sagittal).

fast integration of new modules responsible of different transformations to be applied to the loaded volume, thus facilitating the development of new visualization techniques.

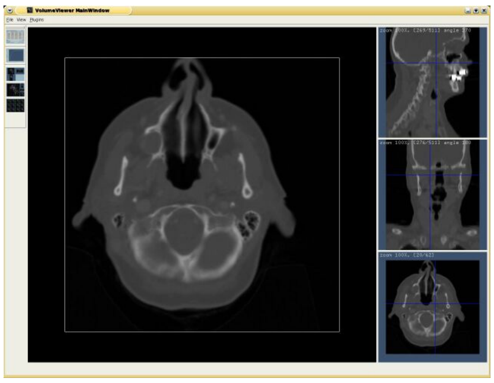
A.1.2.2 Project Partners

The departments of neurosurgery and maxillofacial surgery of the University Hospital Tübingen as well as the BrainLAB company are collaborators of the WSI/GRIS in project VIRTUE.

The project VIRTUE is supported by the German Research Foundation (DFG) in the focus program on “Medical Robotics and Navigation” (SPP 1124).

A.2 FIRE: Isosurface Extraction Software

This second group of projects shows practical examples of applications where the main interest is focused on isosurface extraction and visualization. The two cases here presented are based on variations of our own software framework for isosurface extraction and visualization, called FIRE (**F**ast **I**sosurface **R**endering and **E**xtraction).


(a) Original dataset

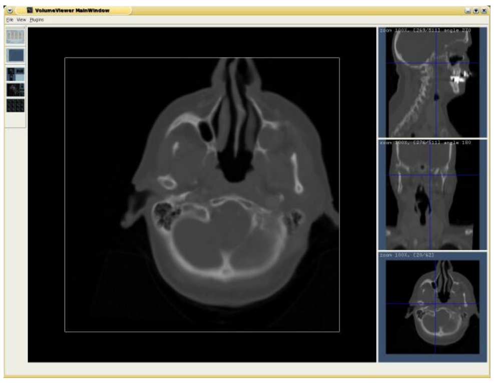

(b) Mirrored dataset

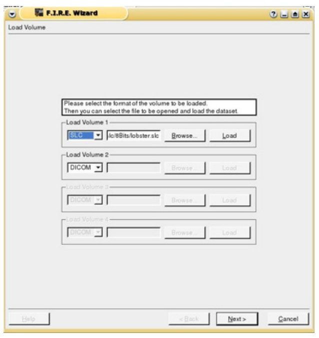
Figure A.5: Example of plug-in in *Helios*/*Volv*.

A.2.1 Extraction of Dinosaur Osseous Structures

A.2.1.1 Description of the Project

The extraction of three-dimensional models from volumetric data is a highly demanded feature that facilitates a successful visualization of these otherwise rather complex data. Usually the region of interest that must be extracted as a 3D model corresponds to the interface of a material within scanned data (from CT or MRI scanners). This isosurface extraction process can be easily carried out employing the Marching Cubes algorithm (see Chapter 3 for a detailed description). By applying the *Marching Cubes* algorithm, a triangulated surface is generated which approximates the shape of a level-set corresponding to an intensity present in the volume. Once an isosurface has been generated, the next logical step is to export this geometrical model into a standard format. This way the 3D model can be stored for further processing with a professional 3D modelling tool such as 3D Studio Max or Maya. All these features, from loading the volume, to extracting the isosurface and exporting to a standard 3D model format, have been implemented in the FIRE software system (see Figure A.6) with a wizard structure that guides the user through the different necessary steps.

The goal in this project is the generation of 3D models of dinosaur bones by extracting isosurfaces from CT-scanned data. Real dinosaur bones are scanned by means of a CT-scanner and the resulting volumetric data is stored as DICOM files. These files are then loaded into FIRE. There, a proper iso-value for the isosurface extraction is interactively determined in a short trial and error process. Once the adequate isosurface has been found and

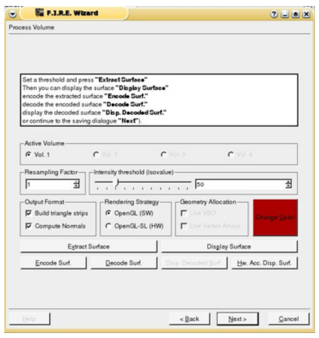


Load Volume

Please select the format of the volume to be loaded.
Then you can select the file to be opened and load the dataset.

Load Volume 1: SLC, lc/8Bits/lobster.slc
Load Volume 2: DICOM
Load Volume 3: DICOM
Load Volume 4: DICOM

(a) Load volume

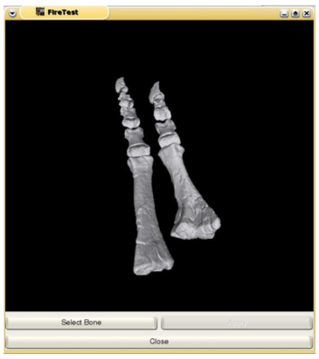


Process Volume

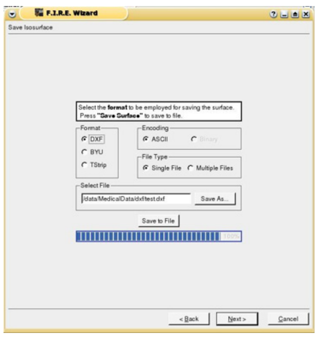
Set a threshold and press **"Extract Surface"**
Then you can display the surface **"Display Surface"**
encode the extracted surface **"Encode Surf."**
decode the encoded surface **"Decode Surf."**
display the decoded surface **"Disp. Decoded Surf."**
or continue to the saving dialogue **"Next"**).

Active Volume: [✓ Vol. 1] Vol. 2 Vol. 3 Vol. 4
Resampling Factor: 1
Intensity threshold (isovalue): 50
Output Format: [✓ Build triangle strips] [✓ Compute Normals]
Rendering Strategy: [✓ OpenGL (SW)] OpenGL-SL (HW)
Geometry Allocation: Use VBO, Use Vertex Arrays

(b) Extract isosurface



(c) Display isosurface



Save Isosurface

Select the **format** to be employed for saving the surface.
Press **"Save Surface"** to save to file.

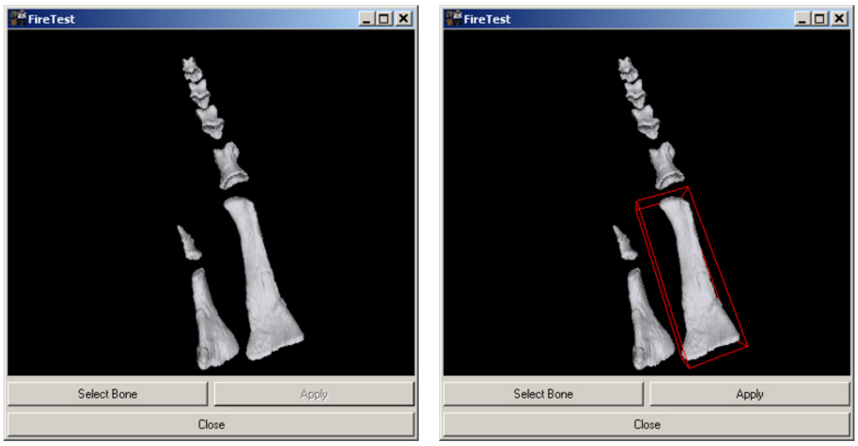
Format: [✓ DXF] BYU TStrip
Encoding: [✓ ASCII] Binary
File Type: [✓ Single File] Multiple Files
Select File: /data/MedicalData/dxf/test.dxf
100%

(d) Export isosurface

Figure A.6: FIRE software system.

extracted, it can be further exported into a file in a 3D model format such as the DXF format, which is compatible with most commercial modelling software.

An additional feature has been implemented in order to deal with those cases where several bones have been scanned simultaneously in the same acquisition and therefore are stored in a single volume. If a standard application of the FIRE pipeline were employed in such case, different bone elements would be stored as a whole in a single 3D model. For the further utilization of the extracted bone models, separate files are required. Therefore, the possibility of choosing a single element out of a compound scene with several bones has been implemented. This way, the user selects an element on the displayed surface and the selected point is used as a seed point for a region growing algorithm that identifies the isosurface of interest. One illustrative example of this situation can be seen in Figure A.7.



(a) Compound scene (b) Selected bone

Figure A.7: Visualization of two toes of a *plateosaurus* showing the extracted bones and the result after selecting one single phalange to be exported.

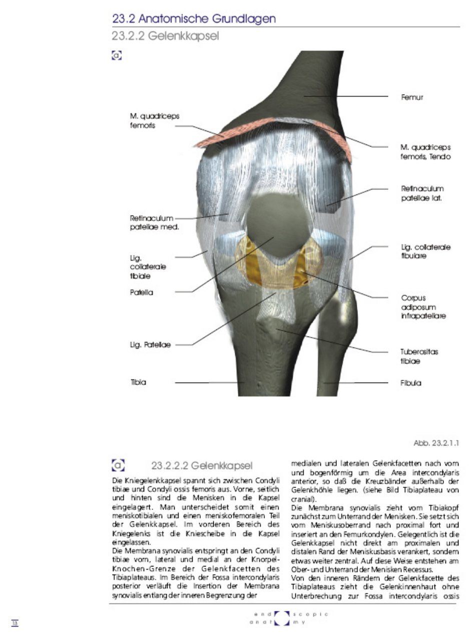
A.2.1.2 Project Partners

This project is the result of an active cooperation between the GRIS department and Prof. H.-U. Pfretzschner and Heinrich Mallison, from the group of *Wirbeltierpaläontologie* (Vertebrate Paleontology) at the University of Tübingen.

A.2.2 Generation and Visualization of Anatomical Models

A.2.2.1 Description of the Project

This project is also based on the isosurface extraction with the *Marching Cubes* algorithm integrated in the FIRE software system. In this case, the user-friendly interactive wizard structure is employed to generate 3D models from human anatomy. The procedure is similar to that of the previous project. Here, instead of dinosaur bones, plastic models resembling the shape of human bones are put into a CT scanner. The corresponding isosurface is subsequently extracted with our software and exported to a file in a format compatible with commercial 3D modelling software. These 3D models are then further processed and integrated into an illustration for an endoscopic anatomy atlas, which is the final objective of the project. Figure A.8 shows two example pages of the atlas that were generated with 3D models extracted



Femur
M. quadriceps femoris
M. quadriceps femoris Tendo
Retinaculum patellae lat.
Retinaculum patellae med.
Lig. collaterale tibiale
Lig. collaterale fibulare
Patella
Corpus adiposum infrapatellare
Lig. Patellae
Tuberositas tibiae
Tibia
Fibula
e n d o s c o p i c a n a t o m y

23.2 Anatomische Grundlagen

23.2.2 Gelenkkapsel

Abb. 23.2.1.1

23.2.2.2 Gelenkkapsel

Die Kniegelenkkapsel spannt sich zwischen Condyli tibiae und Condyli ossis femoris aus. Vorne, seitlich und hinten sind die Menisken in die Kapsel eingelagert. Man unterscheidet somit einen meniskotibialen und einen meniskofemoralen Teil der Gelenkkapsel. Im vorderen Bereich des Kniegelenks ist die Kniescheibe in die Kapsel eingelassen.
Die Membrana synovialis entspringt an den Condyli tibiae vorn, lateral und medial an der Knorpel-Knochen-Grenze der Gelenkfacetten des Tibiaplateaus. Im Bereich der Fossa intercondylaris posterior verläuft die Insertion der Membrana synovialis entlang der inneren Begrenzung der medialen und lateralen Gelenkfacetten nach vorn und bogenförmig um die Area intercondylaris anterior, so daß die Kreuzbänder außerhalb der Gelenkhöhle liegen. (siehe Bild Tibiaplateau von cranial).
Die Membrana synovialis zieht vom Tibiakopf zunächst zum Unterrand der Menisken. Sie setzt sich vom Meniskusoberrand nach proximal fort und inseriert an den Femurkondylen. Gelegentlich ist die Gelenkkapsel nicht direkt am proximalen und distalen Rand der Meniskusbasis verankert, sondern etwas weiter zentral. Auf diese Weise entstehen am Ober- und Unterrand der Menisken Recessus.
Von den inneren Rändern der Gelenkfacette des Tibiaplateaus zieht die Gelenkinnenhaut ohne Unterbrechung zur Fossa intercondylaris ossis

(a)

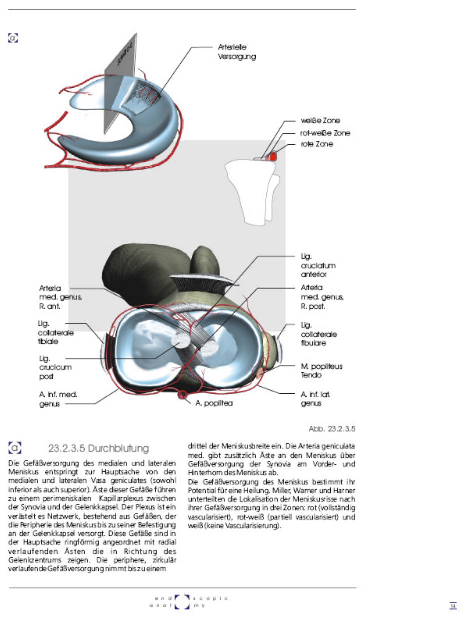


Abb. 23.2.3.5

23.2.3.5 Durchblutung

Die Gefäßversorgung des medialen und lateralen Meniskus entspringt zur Hauptsache von den medialen und lateralen Vasa geniculates (sowohl inferior als auch superior). Äste dieser Gefäße führen zu einem perimeniskalen Kapillarplexus zwischen der Synovia und der Gelenkkapsel. Der Plexus ist ein verästelt es Netzwerk, bestehend aus Gefäßen, der die Peripherie des Meniskus bis zu seiner Befestigung an der Gelenkkapsel versorgt. Diese Gefäße sind in der Hauptsache ringförmig angeordnet mit radial verlaufenden Ästen die in Richtung des Gelenkzentrums zeigen. Die periphere, zirkulär verlaufende Gefäßversorgung nimmt bis zu einem drittel der Meniskusbreite ein. Die Arteria geniculata med. gibt zusätzlich Äste an den Meniskus über Gefäßversorgung der Synovia am Vorder- und Hinterhorn des Meniskus ab.
Die Gefäßversorgung des Meniskus bestimmt ihr Potential für eine Heilung. Miller, Warner und Harner unterteilten die Lokalisation der Meniskusrisse nach ihrer Gefäßversorgung in drei Zonen: rot (vollständig vascularisiert), rot-weiß (partiell vascularisiert) und weiß (keine Vascularisierung).

(b)

Figure A.8: Example pages of the *Endoscopic Anatomy Atlas.*

with FIRE.

A.2.2.2 Project Partners

This project is the result of joint work between the GRIS department and the group of *Endoscopic Anatomy* at the University of Tübingen.

APPENDIX B

Shader Programs

B.1 Vertex Program

The following OpenGL Shading Language (GLSL) vertex program is employed for decoding isosurface vertices as described in Chapter 3. The program computes the texture coordinates to be used within the fragment program in order to access the vertex information and applies an orthographic projection by multiplying each coordinate of a textured quad by the current *modelview* matrix.

```
void main(void)
{
  gl_TexCoord[0] = gl_MultiTexCoord0;
  gl_Position = ftransform();
}
```

B.2 Fragment Program

The following OpenGL Shading Language (GLSL) fragment program is employed for decoding isosurface vertices as described in Chapter 3. This program reads the encoded version of the current vertex from texture memory and applies the necessary transformations in order to compute its actual position and normal vector coordinates. These are then written to two different off-screen buffers (`gl_FragData[0]` and `gl_FragData[1]`). These buffers are then utilized as the source of a vertex array within the application to render the decoded geometry.

```
uniform vec3 volSpacing;
uniform vec3 volOffset;

uniform sampler2D tex0; // ijktTexture
uniform sampler2D tex1; // edgeIdTexture
uniform sampler2D tex2; // normalTexture

const float PI_VAL = 3.14159265358979323846;

void main (void)
{
  vec4 data0 = texture2D(tex0, gl_TexCoord[0].st);
  vec3 voxelIndx = floor(255.0 * data0.xyz + 0.5);

  float edgeID = floor(255.0
                       * texture2D(tex1,
                                   gl_TexCoord[0].st).w
                       + 0.5);

  // Decode the position of the vertex
  vec3 firstVoxelCoord = volOffset +
                         volSpacing * voxelIndx;
  vec3 secondVoxelCoord = firstVoxelCoord + volSpacing;
  vec3 mixed = mix(firstVoxelCoord,
                   secondVoxelCoord,
                   data0.w);

  int2 decision = step(vec2(0.5, 1.5),
                       vec2(edgeID, edgeID));
  gl_FragData[0] = mix(vec4(mixed.x,
                                      firstVoxelCoord.yz,
                                      edgeID),
                                mix(vec4(firstVoxelCoord.x,
                                         mixed.y,
                                         firstVoxelCoord.z,
                                         edgeID),
                                    vec4(firstVoxelCoord.xy,
                                         mixed.z,
                                         edgeID),
                                    decision[1]),
                                decision[0]);

  // Decode the normal vector
  vec4 normalPolar = floor(255.0
                           * texture2D(tex2,
                                       gl_TexCoord[0].st)
                           + 0.5);
  float theta = normalPolar.x * 2.0 * PI_VAL / 255.0;
  float phi = normalPolar.w * PI_VAL / 255.0;
  gl_FragData[1] = vec4(-cos(theta) * sin(phi),
                        -sin(theta) * sin(phi),
                         cos(phi),
                         edgeID);
}
```

Authored or Co-Authored Publications

Articles in Conference Proceedings and Journals

[1] M. Amor, M. Bóo, A. del Río, M. Wand, and W. Straßer. A New Algorithm for High-Speed Projection in Point Rendering Applications. In *DSD-Euromicro Conference. IEEE Computer Society*, pages 337–345, 2003.

[2] D. Bartz, J. Fischer, A. del Río, J. Hoffmann, and D. Freudenstein. VIRTUE: A Navigated Virtual Endoscopy System for Maxillo-Facial and Neurosurgery. In *3D Modelling*, 2003.

[3] A. del Río, M. Amor, M. Bóo, and X. M. Pardo. Visualization of Anatomical Structures Based on Implicit Surfaces and Displacement Mapping. In *WSCG Poster Papers Proceedings*, pages 45–48, 2004.

[4] A. del Río, D. Bartz, R. Jäger, O. Gürvit, and D. Freudenstein. Efficient Stereoscopic Rendering in Virtual Endoscopy Applications. In *Proc. of WSCG*, pages 95–101, 2003.

[5] A. del Río, M. Bóo, M. Amor, and J. D. Bruguera. Hardware Implementation of the Subdivision Loop Algorithm. *Proc. of 28th Euromicro Conference. IEEE Computer Society*, pages 189–197, September 2002.

[6] A. del Río, J. Fischer, D. Bartz, and W. Straßer. Fast Rendering of Large Encoded Isosurfaces from Uniform Grid Datasets. In *Vision, Modeling, and Visualization (VMV)*, pages 71–78, 2005.

[7] A. del Río, J. Fischer, M. Köbele, D. Bartz, J. Hoffmann, F. Duffner, M. Tatagiba, and W. Straßer. Intuitive Volume Classification in Medical AR. In *Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e. V. (CURAC)*, 2005.

[8] A. del Río, J. Fischer, M. Köbele, D. Bartz, and W. Straßer. Augmented Reality Interaction for Semiautomatic Volume Classification. In E. K. R. Blach, editor, *Eurographics Workshop on Virtual Environments (EGVE)*, pages 113–120, 2005.

[9] A. del Río, Z. Salah, D. Bartz, F. Freudenstein, and F. Dammann. Enhanced Segmentation and Visualization of Blood Vessels in CT-Angiography Applications. In *Jahrestagung der Deutschen Gesellschaft für Computer-und Roboterassistierte Chirurgie e.V.(CURAC)*, 2004.

[10] J. Fischer, D. Bartz, A. del Río, Z. Salah, J. Orman, D. Freudenstein, J. Hoffmann, and W. Straßer. VIRTUE: Ein System der navigierten virtuellen Endoskopie für die MKG- und Neurochirurgie (Poster). In *Gemeinsame Jahrestagung der Österreichischen, Deutschen und Schweizerischen Gesellschaft für Biomedizinische Technik*, 2003.

[11] J. Fischer and A. del Río. A Fast Method for applying Rigid Transformations to Volume Data. In *Proc. of WSCG*, pages 55–62, 2004.

[12] J. Fischer, A. del Río, M. Mekic, D. Bartz, J. Hoffmann, and W. Strasser. Effizientes Spiegeln von Volumendaten an einer beliebigen Ebene für die MKG-Chirurgie. In *Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e. V.(CURAC)*, November 2003.

[13] J. Fischer, M. Mekic, A. del Río, D. Bartz, and J. Hoffmann. Prototyping a Planning System for Orbital Reconstruction. In *Proc. of Workshop Bildverarbeitung für die Medizin*, pages 264–268, March 2004.

[14] J. Hoffmann, D. Troitzsch, C. Westendorff, D. Bartz, J. Fischer, A. del Río, F. Dammann, and S. Reinert. Bilddatengestützte präoperative Planung und intraoperative navigationsgestützte Rekonstrution von Orbitawandfrakturen. In *38. DGBMT Jahrestagung Biomedizinische Technik (BMT)*, 2004.

[15] J. Hoffmann, D. Troitzsch, C. Westendorff, A. del Río, J. Fischer, D. Bartz, and S. Reinert. Bilddatengestützte Navigation und optische Endoskopie zur minimal invasiven kraniomaxillofazialen

Chirurgie (Poster). In *Gemeinsame Jahrestagung der Österreichischen, Deutschen und Schweizerischen Gesellschaft für Biomedizinische Technik*, 2003.

[16] J. Hoffmann, C. Westendorff, D. Bartz, J. Fischer, A. del Rio, and S. Reinert. Computergestützte planung und navigationsgestütztes operatives vorgehen bei komplexen orbitawandeffekten. In *Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e. V. (CURAC)*, 2005.

[17] D. Mayer, D. Bartz, J. Fischer, S. Ley, A. del Río, S. Thust, H. Kauczor, W. Straßer, and C. Heussel. Hybrid Segmentation and Virtual Bronchoscopy Based on CT Images. *Academic Radiology*, 11(5):551–565, 2004.

[18] C. Westendorff, D. Gülicher, J. Fischer, D. Bartz, A. del Rio, S. Reinert, and J. Hoffmann. Computergestützte planung und ct-datenbasierte navigation zur rekonstruktion des lateralen mittelgesichts bei jochbeinfehlstellungen. In *Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e. V. (CURAC)*, 2005.

[19] C. Westendorff, J. Kaminsky, A. del Rio, J. Fischer, M. Tatagiba, S. Reinert, and J. Hoffmann. Resektion eines ausgedehnten keilbeinflühelmeningeoms mit infiltration der orbita unter gebrauch computergestützter planung und cad/cam technologie zur rekontruktion. In *Jahrestagung der Deutschen Gesellschaft für Computer- und Roboterassistierte Chirurgie e. V. (CURAC)*, 2005.

[20] G. Wetekam and A. del Río. Visualization and User Interface for Multi-Slice CT Fluoroscopy. In *Tagungsband Fachtagung Biomedizinische Technik (BMT)*, 38. DGBMT Jahrestagung, September 2004.

Technical Reports

[21] D. Bartz, B. Schnaidt, J. Cernik, L. Gauckler, J. Fischer, and A. del Río. Volumetric High Dynamic Range Windowing for Better Data Representation. Technical Report WSI-2005-03, Wilhelm Schickard Institute for Computer Science, Graphical-Interactive Systems (WSI/-GRIS), University of Tübingen, January 2005.

Bibliography

[22] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS 1968 Spring Joint Computer Conference*, pages 37–45, 1968.

[23] R. Azuma. A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[24] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast Isocontouring for Improved Interactivity. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 39–46., 1996.

[25] C. L. Bajaj, V. Pascucci, and D. R. Schikore. The Contour Spectrum. In *Proceedings of IEEE Visualization*, pages 167–173., 1997.

[26] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[27] M. Bentum. *Interactive Visualization of Volume Data.* PhD thesis, University of Twente, Enschede, The Netherlands, June 1995.

[28] F. Bernardini, H. Rushmeier, I. M. Martin, J. Mittleman, and G. Taubin. Building a Digital Model of Michelangelo's Florentine Pietà. *IEEE Computer Graphichs and Applications*, 22(1):59–67, 2002.

[29] J. Bloomenthal. *Introduction to Implicit Surfaces (The Morgan Kaufmann Series in Computer Graphics).* Morgan Kaufmann Publishers, Inc., 1997.

[30] M. Botsch and L. Kobbelt. High-Quality Point-Based Rendering on Modern GPUs. In *Proceedings of the Pacific Conference on Computer Graphics and Applications (PG)*, pages 335–346, 2003.

[31] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[32] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Report utec-csc-74-133, Computer Science Department, University of Utah, 1974.

[33] M. M. Chow. Optimized Geometric Compression for Real-Time Rendering. In *Proceedings of IEEE Visualization*, pages 347–354, 1997.

[34] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[35] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal Isosurface Extraction from Irregular Volume Data. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 31–38, 1996.

[36] L. Coconu and H.-C. Hege. Hardware-Accelerated Point-Based Rendering of Complex Scenes. In *Proceedings of the Eurographics Workshop on Rendering*, pages 43–52, 2002.

[37] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, and M. Howard. Towards an Interactive High Visual Complexity Animation System. *Proceedings of ACM SIGGRAPH*, 13(2):289–299, 1979.

[38] T. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina at Chapel Hill, 1993.

[39] L. H. de Figueiredo, J. de Miranda Gomes, D. Terzopoulos, and L. Velho. Physically-Based Methods for Polygonization of Implicit Surfaces. In *Proceedings of the Conference on Graphics Interface*, pages 250–257, 1992.

[40] M. Deering. Geometry Compression. In *Proceedings of ACM SIGGRAPH*, pages 13–20, 1995.

[41] A. Doi and A. Koide. An Efficient Method of Triangulating Equi-valued Surfaces by Using Tetrahedral Cells. *IEICE Transactions on Communications, Electronics, Information and Systems*, E-74(1):214–224, 1991.

[42] R. A. Drebin. Personal Communication. ATI, 1992.

[43] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Proceedings of ACM SIGGRAPH*, pages 65–74, 1988.

[44] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-Time Volume Graphics - Course Notes #28. In *Proceedings of ACM SIGGRAPH*, 2004.

[45] M. Figl, W. Birkfellner, J. Hummel, R. Hanel, P. Homolka, F. Watzinger, F. Wanschitz, R. Ewers, and H. Bergmann. Current Status of the Varioscope AR, a Head-Mounted Operating Microscope for Computer-Aided Surgery. In *Proceedings of IEEE International Symposium on Augmented Reality (ISAR)*, pages 20–29, October 2001.

[46] J. Fischer and D. Bartz. Utilizing Image Guided Surgery for User Interaction in Medical Augmented Reality. Technical Report WSI-2005-04, WSI/GRIS, University of Tübingen, March 2005.

[47] J. Fischer, M. Neff, D. Freudenstein, and D. Bartz. Medical Augmented Reality Based on Commercial Image Guided Surgery. In *Proceedings of Eurographics Symposium on Virtual Environments (EGVE)*, pages 83–86, June 2004.

[48] J. Fischer, M. Neff, D. Freudenstein, D. Bartz, and W. Straßer. ARGUS: Harnessing Intraoperative Navigation for Augmented Reality. In *Proceedings of the DGBMT Jahrestagung Biomedizinische Technik (BMT)*, pages 42–43, September 2004.

[49] K. G. Freeman. Technique and System for the Real-Time Generation of Perspective Images. U.S. Patent, 1996. Number 5,550,959.

[50] J. Grossman. *Point Sample Rendering.* Master's thesis, Dept. of Electrical Engineering and Computer Science, Stanford University, 1998.

[51] J. P. Grossman and W. J. Dally. Point Sample Rendering. In *Rendering Techniques (Eurographics)*, pages 181–192. Springer-Verlag Wien New York, 1998.

[52] S. Gumhold and W. Straßer. Real Time Compression of Triangle Mesh Connectivity. In *Proceedings of ACM SIGGRAPH*, pages 133–140, 1998.

[53] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization*, pages 53–59, 2002.

[54] T. He, L. Hong, A. Kaufman, and H. Pfister. Generation of Transfer Functions with Stochastic Search Techniques. In *Proceedings of IEEE Visualization*, pages 227–237., 1996.

[55] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphichs and Applications*, 6(11):56–67, 1986.

[56] H. Hoppe. Smooth View-Dependent Level-Of-Detail Control and its Application to Terrain Rendering. In *Proceedings of IEEE Visualization*, pages 35–42, 1998.

[57] N. Ide, M. Hirano, Y. Endo, S. Yoshioka, H. Murakami, A. Kunimatsu, T. Sato, T. Kamei, T. Okada, and M. Suzuoki. 2.44-GFLOPS 300-MHz Floating-Point Vector-Processing Unit for High-Performance 3-D Graphics Computing. *IEEE Journal of Solid-State Circuits*, 35(7):1025–1033, July 2000.

[58] M. Isenburg, P. Lindstrom, S. Gumhold, and J. Snoeyink. Large Mesh Simplification using Processing Sequences. In *Proceedings of IEEE Visualization*, pages 465–472, 2003.

[59] T. Itoh, Y. Yamaguchi, and K. Koyamada. Automatic Isosurface Propagation by Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.

[60] T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume Thinning for Automatic Isosurface Propagation. In *Proceedings of IEEE Visualization*, pages 303–310., 1996.

[61] T. Itoh, Y. Yamaguchi, and K. Koyamada. Fast Isosurface Generation Using the Volume Thinning Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):32–46, 2001.

[62] U. Kanus, G. Wetekam, J. Hirche, and M. Meißner. VIZARDII: An FPGA-based Interactive Volume Rendering System. In *Proceedings of Field-Programmable Logic and Applications*, pages 1114–1117, 2002.

[63] G. Kindlmann and J. W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 79–86, 1998.

[64] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of the Pacific Conference on Computer Graphics and Applications (PG)*, pages 186–195, 2004.

[65] J. Kniss, G. Kindlmann, and C. Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proceedings of IEEE Visualization*, pages 255–262, 2001.

[66] H. Kobayashi, T. Maeda, H. Yamauchi, and T. Nakamura. A Cached Frame Buffer System for Object-Space Parallel Processing Systems. In *Proceedings of Computer Graphics International (CGI)*, pages 146–155, 1997.

[67] M. Köbele. Augmented-Reality Based Classification for Volume Rendering in Medical Visualisation. Diplomarbeit, University of Tübingen, April 2005.

[68] A. H. König and E. M. Gröller. Mastering Transfer Function Specification by Using VolumePro Technology. In *Proceedings of Spring Conference on Computer Graphics (SCCG)*, pages 279–286, April 2001.

[69] J. Krüger, J. Schneider, and R. Westermann. DuoDecim - A Structure for Point Scan Compression and Rendering. In *Proceedings of Eurographics Symposium on Point-Based Graphics*, 2005.

[70] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization*, pages 287–292, 2003.

[71] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proceedings of ACM SIGGRAPH*, pages 451–458, 1994.

[72] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proceedings of ACM SIGGRAPH*, pages 285–288, 1991.

[73] D. T. Lee and C. K. Wong. Worst Case Analysis for Region and Partial Region Searches in Multi-Dimensional Binary Search Trees and Balanced Quad Trees. *Acta Informatica*, 9(23):23–29, 1977.

[74] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphichs and Applications*, 8(3):29–37, 1988.

[75] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Proceedings of ACM SIGGRAPH*, pages 131–144, 2000.

[76] M. Levoy and T. Whitted. The Use of Points as a Display Primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[77] D. Lischinski and A. Rappoport. Image-Based Rendering for Non-Diffuse Synthetic Scenes. *Rendering Techniques (Eurographics)*, pages 301–314, June 1998.

[78] Y. Livnat. *NOISE, WISE and SAGE: Algorithms for Rapid Isosurface Generation*. PhD thesis, Department of Computer Science, 1999.

[79] Y. Livnat. *The Visualization Handbook (Hansen/Johnson ed.)*, chapter 2. Accelerated Isosurface Extraction Approaches, pages 35–52. Academic Press, Dezember 2004.

[80] Y. Livnat and C. Hansen. View-Dependent Iso-Surface Extraction. In *Proceedings of IEEE Visualization*, pages 175–180, 1998.

[81] Y. Livnat, H.-W. Shen, and C. R. Johnson. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[82] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169, 1987.

[83] R. Mace. OpenGL ARB Superbuffers. Available on: `http://www.ati.com/developer/gdc/SuperBuffers.pdf`, 2004.

[84] D. Manocha (Organizer). Interactive Geometric Computations Using Graphics Hardware - Course Notes #31. In *Proceedings of ACM SIGGRAPH*, 2002.

[85] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design Galleries: A General Approach to

Setting Parameters for Computer Graphics and Animation. In *Proceedings of ACM SIGGRAPH*, pages 389–400, 1997.

[86] S. V. Matveyev. Approximation of Isosurface in the Marching Cube: Ambiguity Problem. In *Proceedings of IEEE Visualization*, pages 288–292, 1994.

[87] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[88] B. H. McCormick, T. A. DeFanti, and M. D. Brown. Visualization in Scientific Computing. *Computer Graphics*, 21(6):1–4, 1987.

[89] M. Meißner, U. Kanus, and W. Straßer. VIZARD II, A PCI-Card for Real-Time Volume Rendering. In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 61–68, 1998.

[90] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARDII: A reconfigurable interactive volume rendering system. In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 137–146, 2002.

[91] T. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

[92] K. Mueller and R. Crawfis. Eliminating Popping Artifacts in Sheet Buffer-Based Splatting. In *Proceedings of IEEE Visualization*, pages 239–245, 1998.

[93] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):116–134, 1999.

[94] G. M. Nielson, T. A. Foley, B. Hamann, and D. Lane. Visualizing and Modeling Scattered Multivariate Data. *IEEE Computer Graphichs and Applications*, 11(3):47–55, 1991.

[95] K. M. Oh and K. H. Park. A Type-Merging Algorithm for Extracting an Isosurface from Volumetric Data. *The Visual Computer*, 12:406–419, 1996.

[96] M. Pfeifle. Design and Implementation of a Graphical User Interface (GUI) for Medical Visualization. Studienarbeit, University of Tübingen, 2005.

[97] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *Proceedings of ACM SIGGRAPH*, pages 251–260, 1999.

[98] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. S. Avila, K. Martin, R. Machiraju, and J. Lee. The Transfer Function Bake-Off. *IEEE Computer Graphichs and Applications*, 21(3):16–22, 2001.

[99] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface Elements as Rendering Primitives. In *Proceedings of ACM SIGGRAPH*, pages 335–342, 2000.

[100] V. Popescu, J. Eyles, A. Lastra, J. Steinhurst, N. England, and L. Nyland. The WarpEngine: an Architecture for the Post-Polygonal Age. In *Proceedings of ACM SIGGRAPH*, pages 433–442, 2000.

[101] T. Poston, H. T. Nguyen, P.-A. Heng, and T.-T. Wong. "Skeleton Climbing": Fast Isosurfaces with Fewer Triangles. In *Proceedings of the Pacific Conference on Computer Graphics and Applications (PG)*, pages 117–126, 1997.

[102] W. T. Reeves. Particle Systems – a Technique for Modeling a Class of Fuzzy Objects. *ACM Transactions on Graphics (TOG)*, 2(2):91–108, 1983.

[103] W. T. Reeves and R. Blau. Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. In *Proceedings of ACM SIGGRAPH*, pages 313–322, 1985.

[104] B. Reitinger, C. Zach, A. Bornik, and R. Bichel. User-Centric Transfer Function Specification in Augmented Reality. In *Proceedings of WSCG*, volume 12, Plzen, Czech Republic, February 2004.

[105] L. Ren, H. Pfister, and M. Zwicker. Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. In *Proceedings of IEEE Eurographics*, 2002.

[106] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of Joint EUROGRAPHICS - IEEE TCCG Symposium on Visualization (VisSym*, pages 231–238, 2003.

[107] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In *Proceedings of ACM SIGGRAPH*, pages 381–394, 1994.

[108] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of ACM SIGGRAPH*, pages 343–352, 2000.

[109] P. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. In *Proceedings of ACM SIGGRAPH*, pages 51–58, 1988.

[110] D. Saupe and J.-P. Kuska. Compression of Isosurfaces for Structured Volumes. In *Proceedings of Vision, Modeling and Visualization (VMV)*, pages 333–340, 2001.

[111] D. Schmalstieg, A. Fuhrmann, G. Hesina, Z. Szalavári, L. M. Encarnação, M. Gervautz, and W. Purgathofer. The Studierstube Augmented Reality Project. *Presence: Teleoperators and Virtual Environments*, 11(1):33–54, 2002.

[112] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, Massachusetts, 1992.

[113] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered Depth Images. In *Proceedings of ACM SIGGRAPH*, pages 231–242, 1998.

[114] J. Shade, D. Lischinski, D. H. Salesin, T. DeRose, and J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Proceedings of ACM SIGGRAPH*, pages 75–82, 1996.

[115] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill. Octree-Based Decimation of Marching Cubes Surfaces. In *Proceedings of IEEE Visualization*, pages 335–342, 1996.

[116] L.-J. Shiue, I. Jones, and J. Peters. A Realtime GPU Subdivision Kernel. In *Proceedings of ACM SIGGRAPH*, pages 1010–1015, 2005.

[117] A. R. Smith. Plants, Fractals, and Formal Languages. In *Proceedings of ACM SIGGRAPH*, pages 1–10, 1984.

[118] A. State, M. Livingston, G. Hirota, W. Garrett, M. Whitton, H. Fuchs, and E. Pisano. Technologies for Augmented-Reality Systems: Realizing Ultrasound-Guided Needle Biopsies. In *Proceedings of ACM SIGGRAPH*, pages 439–446, August 1996.

[119] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten.* PhD thesis, Technische Universität Berlin, 1974.

[120] P. Sutton, C. Hansen, H.-W. Shen, and D. Schikore. A Case Study of Isosurface Extraction Algorithm Performance. In *Proceedings of Joint EUROGRAPHICS - IEEE TCCG Symposium on Visualization (VisSym*, pages 259–268, 2000.

[121] R. Szeliski and D. Tonnesen. Surface Modeling with Oriented Particle Systems. In *Proceedings of ACM SIGGRAPH*, pages 185–194, 1992.

[122] G. Taubin. BLIC: Bi-Level Isosurface Compression. In *Proceedings of IEEE Visualization*, pages 451–458, 2002.

[123] G. Taubin and J. Rossignac. Geometrical Compression through Topological Surgery. *ACM Transactions on Graphics (TOG)*, 17:84–115, 1998.

[124] H. K. Tuy and L. T. Tuy. Direct 2-D Display of 3-D Objects. *IEEE Computer Graphichs and Applications*, 4(10):29–33, 1984.

[125] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma. A Novel Interface for Higher-Dimensional Classification of Volume Data. In *Proceedings of IEEE Visualization*, 2003.

[126] F.-Y. Tzeng, E. B. Lum, and K.-L. Ma. An Intelligent System Approach to Higher-Dimensional Classification of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):273–284, May/June 2005.

[127] A. van Gelder and J. Wilhelms. Topological Considerations in Isosurface Generation. *ACM Transactions on Graphics (TOG)*, 13(4):337–375, 1994.

[128] S. Vogt, A. Khamene, F. Sauer, A. Keil, and H. Niemann. A High Performance AR System for Medical Applications. In *Proceedings of IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 270–271, October 2003.

[129] M. Wand. *Point-Based Multi-Resolution Rendering.* PhD thesis, Department of Computer Science and Cognitive Science, University of Tübingen, 2004.

[130] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Strasser. The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Proceedings of ACM SIGGRAPH*, pages 361–370, 2001.

[131] L. Westover. Footprint Evaluation for Volume Rendering. In *Proceedings of ACM SIGGRAPH*, pages 367–376, 1990.

[132] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[133] J. Wilhelms and A. van Gelder. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, 1992.

[134] A. P. Witkin and P. S. Heckbert. Using Particles to Sample and Control Implicit Surfaces. In *Proceedings of ACM SIGGRAPH*, pages 269–277, 1994.

[135] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation for Volume Sampling. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 135–142, 1998.

[136] H. Yamauchi, T. Maeda, H. Kobayashi, and T. Nakamura. The Object-Space Parallel Processing of the Multipass Rendering Method on the $(m\pi)^2$ with a Distributed-Frame Buffer System. *IEICE Transactions on Information and Systems*, E80-D(9):909–918, September 1997.

[137] S.-N. Yang and T.-S. Wu. Compressing Isosurfaces Generated with Marching Cubes. *The Visual Computer*, 18:54–67, 2002.

[138] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface Splatting. In *Proceedings of ACM SIGGRAPH*, pages 371–378, 2001.

Lebens- und Bildungsgang

Ángel del Río Fernández

15. Juli 1978	geboren in Santiago de Compostela (Spanien)
1984 - 1992	*Educación General Básica (E.G.B.)* (Grundschule) Colegio La Salle, Santiago de Compostela (Spanien)
1992 - 1995	*Bachillerato Unificado Polivalente (B.U.P.)* (Gymnasium) I.B. Rosalía de Castro, Santiago de Compostela (Spanien)
1994 - 1996	International Baccalaureate im I.B. Rosalía de Castro, Santiago de Compostela (Spanien)
1995 - 1996	*Curso de Orientación Universitaria (C.O.U.)* mit Auszeichnung im I.B. Rosalía de Castro, Santiago de Compostela (Spanien)
1996	*Premio Extraordinario de Bachillerato*
1996	Abschluß P.A.A.U. (Abitur)
1996 - 2001	*Licenciatura* in Physik an der Universität Santiago de Compostela (Spanien)
2001	*Memoria de Licenciatura* (Diplomarbeit) *Análisis de la Implementación del Algoritmo de Loop para la Subdivisión de Mallas de Triángulos* in Physik an der Universität Santiago de Compostela (Spanien)
10/2001	Abschluss des Studiums als *Licenciado con Grado en Física* (Diplom -Physiker)
seit 11/2001	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Graphisch-Interaktive Systeme von Prof. Dr.-Ing. Dr.-Ing. E.h. Wolfgang Straßer des Wilhelm-Schickard Instituts für Informatik der Universität Tübingen