Kd-Jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs

Hughes, David Meirion

Award date:
2010

Awarding institution:
Bangor University

Link to publication

Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs

David Meirion Hughes

School of Computer Science

Bangor University

A thesis accepted for the degree of

Doctor of Philosophy

July 2010

I dedicate this work, the culmination of two decades of schooling and study, to my parents Wendy and Meirion. Without their support and determination to see me succeed I would not have had the opportunity to go to university. I thank Ruth for being a wonderful sister and a source of inspiration by dedicating her life to a dream and working each day to realise it.

Lots of thanks go my very good friends: Lee, Nigel, Tom, Andy, Jen and Kirsty for keeping me sane over the years. Without them my path through life would be very lonely and I'm very privileged to have them as friends.

I also wish to thank my partner Tracy for helping me pass the final hurdle to complete my PhD. The final months were the most stressful and her support was at a time when it mattered the most.

Acknowledgements

I would like to acknowledge my supervisor Dr Ik Soo Lim for his support and patience during my PhD research. I also acknowledge and thank Prof. Nigel John and Dr Ik Soo Lim for giving me the opportunity to study at Bangor University.

I would also like to acknowledge my office friends; James, Rhys, Catrin and Tom for making our office an enjoyable and friendly place to work in.

Abstract

Advances in Graphical Processing Units (GPUs) bring both opportunities and challenges for the acceleration of volumetric visualisation. Many researchers have highlighted these problems, such as GPU memory access and low computation-throughput bottlenecks. For raytracing, stackless traversal techniques are often used to circumvent memory bottlenecks by avoiding the use of a stack and instead replacing return traversal with extra computation. This thesis addresses whether the stackless traversal approaches are useful on newer hardware and technology (such as CUDA). In addition this work explores the possibility of accelerating volumetric segmentation to allow real-time user interaction. As segmentation methods, especially those based on machine-learning, will typically examine all the voxels of a volume, they are ideally suited to the parallel nature of GPU computation. Further, context-preserving volume rendering is explored for segmentation data in order to see if rendering the segmentation information of a volume can be useful for users.

In order to explore usefulness of stackless traversal on modern GPUs this thesis presents a novel stackless approach called kd-jump. Kd-jump exploits the benefits of index-based node traversal and formulates a return mechanism based on applying an inverse. Kd-Jump allows traversal to immediately return to the next valid node, when required, without having to backtrack one node at a time or perform additional node testing, as the case is with Kd-Backtrack. This allows kd-jump to avoid incurring extra node visitation, which will typically incur a greater amount of redundant work. The stackless method is achieved by the addition of a single 32-bit integer, which is stored within a fast GPU register, and an accumulation matrix that is stored in constant memory. In addition ray clipping to the

bounds of a node is required upon return. It is shown that Kd-Jump outperforms a stack-based approach by an average range of 10% to 20%.

This thesis presents a context-preserving visualization method for segmentation data derived from volumetric medical images. A segmented volumetric image contains a number of anatomical objects which are important features to be visualized. The context preserving rendering algorithm utilizes the curvature at the surfaces of the segmentation objects to modulate the opacity contribution during rendering. This results in the areas of high curvature, typically the most important features, being opaque and visible and everything else being transparent.

A segmentation tool utilizing support vector machines (SVM) is also presented. This segmentation tool utilizes incremental SVM to allow for realtime learning and unlearning of input data and background training. This enables the SVM to train on previous input while the user continues to provide further input. The theory for such a system is that the complex task of training an SVM does not incur a noticeable delay, specifically after the user has finished inputting data and requires the results. Further, in order to expedite the class prediction of the remaining volume, using the trained SVM, GPUs are employed. CUDA-kernels are utilized to predict the class of each volume voxel and then store the result in a class volume. This first entails transposing the voxel into the higher dimensional space used in the SVM and then computing the weighted-kernel sum. These tasks are completely parallel in respect to the voxels and are perfectly suited for GPU acceleration.

The main contributions of this thesis can be summarized as the following;

- A novel stackless traversal approach for balanced, or left-balanced binary trees, which provides a theorized and proven performance improvement compared to a stack-based approach.
- A volume visualisation method for context-preservation specifically tailored for segmentation data.

- A segmentation tool, which utilizes incremental SVM training, fast GPU-based volume prediction and fast GPU-based segmentation rendering, to enable a user to segment volumetric data in real-time with fewer delays.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the advent of imaging technology and its digital storage much research has been dedicated to creating and improving new ways to analyze such data. Imagining technology has many forms; a digital camera records the reflected light to produce a two-dimensional image of the subject object(s). Computed tomography records multiple X-ray images or the subject and reconstructs a three-dimensional volume reconstruction. Regardless of the input source, the data requires a method for analysis. Analysis can be achieved by many methods; however, the typical solution is to simply reproduce the input such that is can be displayed on a visualization device, such as a computer monitor.

In the case of a two-dimensional image, the data is easily projected onto a monitor with very little effort. However, with volumetric data, the three-dimensional space must be projected onto the two-dimensional screen. Such a problem is not trivial and there are many different methods which produce (at times) vastly different outcomes.

Medical professionals often work and think in terms of 3D space; for example, diagnostic examinations have to take into account not only the size and the shape of

Figure 1.1: Doctor examining patient CT scans

pathologies of interest, but also their spatial position and vicinity to other anatomical structures. Hence, 3D visualization techniques have huge potential in aiding medical professionals to better understand patient anatomy and pathology.

The two main methods for the visualization of volumetric data are direct-volume rendering and isosurface visualization. For both of these visualization approaches there exist numerous algorithms to produce the end result; for example, slice-based projection, surface splatting, and raytracing.

Apart from the visualization of data, other methods for analysis attempt to segment the recorded data into separate regions. Typically for medical data, segmentation means separating organs from other organs. Combining the output of a segmented volume and using it during visualization allows specific regions to be rendered uniquely, or even completely made hidden. These techniques allow for better understanding of how an object relates to its surroundings, or simply to allow a user to observe the object without obstruction.

Segmentation of volumetric data has a vast research following, not least because of the importance of the work for medical applications and practitioners. However, producing an interface which is simple and desirable for use is again not a trivial problem.

1.1 Problem Description

Producing a GUI for the segmentation of volumetric data, and the visualization of results therein, has a number of challenges. Firstly the software must provide a simple, yet robust interface such that new users can easily use it. On the other-hand experienced users should be able to acquire the results they desire quickly. Secondly, the underlying computations performed by the software must not impede the usability of the program; i.e. the user is does not have to wait long periods of time for results, nor should the user witness delays while providing the input data used for directing the segmentation.

There are two features of segmentation software which can suffer from the aforementioned problems. The segmentation computation itself can be slow, especially if machine learning methods are utilized. Not only would machine learning-based segmentations require some form of *training* the entire volume would also need to be *classified* in order to arrive at the complete segmentation. Secondly, visualization of the segmentation data must be fast and useful to the user.

To summarise, these problems are;

- Slow segmentation, where the user must wait for results.
- Slow visualization, where the users experience interacting with the visualization is slow and frustrating.
- Difficulty in understanding the context of a segmentation in relation to the surrounding features.

1.2 Hypothesis

This thesis hypothesizes that with the utilization of Graphical Processing Units, volume visualization can be performed faster by catering to the strengths of GPUs while also avoiding their weaknesses. Specifically, avoiding access to GPU memory and favouring computation can yield faster results for ray tracing. Further, that segmentation of volumetric data can be made faster by utilizing the power of GPUs. Finally, that segmentation data can be directly visualized using context preserving rendering.

1.3 Objectives

In order to prove the hypothesis this thesis will explore the three areas of research and present new work. These objectives are;

- To research the field of segmentation and volume visualization
- To provide a visualization method, which is not only fast and robust, but also tailored to avoid GPU weaknesses.
- To develop a segmentation tool, which is fast yet robust enough to cope with complex segmentations.
- To develop visualization methods for segmentation data, which preserve context and improve understanding of data.

1.4 Papers

The following are the published (or accepted for publication) research papers relevant to this thesis.

D. M. Hughes, I. S. Lim. "Kd-Jump: a Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs". IEEE Transactions on Visualization and Computer Graphics, 15(6), 2009, pp 1555–1562.

D.M. Hughes, I.S. Lim, "Context-Preserving Rendering of Medical Segmentation Data", Proc of 29th International Conference of the IEEE Engineering in Medicine and Biology Society, August 2007, pp. 5521-5524.

D.M. Hughes, I.S. Lim, "A case-study of inconsistent surface reconstruction in recent literature resulting from Octree rotation-variance", Proc. of Theory and Practice of Computer Graphics 2007, Bangor, pp. 195-200.

1.5 Thesis Format

Chapters are organized as follows:

- Chapter 2 outlines the background techniques in the fields of visualization and segmentation.
- Chapter 3 outlines the previous research most relevant to the work contained in this thesis.
- Chapter 4 outlines isosurface and volume rendering approach utilizing an implicit kd-tree specifically tailored for use on GPUs. This chapter also introduces a stackless traversal approach for balanced binary-trees, which swaps memory access for computation.

- Chapter 5 presents a segmentation tool using incremental SVM powered by multiple cores and GPUs. Also presented is a context-preserving rendering technique for segmentation data.
- Chapter 6 presents a method for consistent surface-reconstruction when using methods derived from object-space subdivision; such as an Octree.
- Chapter 7 finalizes with the conclusions of the thesis.

Chapter 2

Background

This chapter outlines the history of volume rendering and the methods developed; from the initial research through to the modern methods being employed today.

2.1 Volumetric Visualization

With the advent of the digital age, many forms of three-dimensional data-sets began to be recorded. The most common was scanned data from Computed Tomography (CT) or Magnetic Resonance Imaging (MRI), and scientific simulations such as fluid flow. These three-dimensional data-sets are formed from two-dimensional slices and can hold information on whatever has been scanned; be it humans, animals of inanimate objects. The main problem was visualizing the data so that it could be analyzed.

Volume visualization is the process of representing a given volume, whether it is three-dimensional or four-dimensional (animated), such that a human can observe it in a meaningful manner. There are two principle disciplines in volume visualization; realistic rendering [333, 281, 126] and scientific visualization (or non-photorealistic rendering) [113, 199, 87]. Realistic rendering attempts to recreate the visual look of something as it is in real life, such as how light interacts with materials, while scientific visualization simply tries to give a visual insight into the data with more emphases on portraying the information.

2.1.1 Surface Contouring

Early volume visualization during the 1960s and 1970s was limited by the computational power available at the time. The simple solution for viewing three-dimensional data was to examine it one slice at a time. However, examining volume slices one at a time does not easily allow an observer to recreate the three-dimensional look and feel in their imagination; in addition it does not enable a doctor to analyze anatomy from all angles. As a result, researchers began to find methods of rendering the data in three-dimensions, from arbitrary view points.

Initial research into (general) computer graphics took the form of vector rasterization; whereby dots, lines and curves are rendered to a two-dimensional buffer, which was then viewed on a display, or printed with a plotter [13]. As vector rasterization was well established, at the time, the initial visualization of volumetric data exploited these techniques; specifically, surface contouring or isosurfacing. Surface contouring is the process of finding and extracting a contour, or multiple contours, from a volumetric data-set and then rasterizing it; [330, 213, 329, 304]. This form of visualization was not ideal because of problems in interpreting the output, as noted by Stevens [291].

The next research step was to polygonize the surface contours and visualize the resulting triangle mesh. Early work in this field attempted to connect contours of a slice with the contours of the next; for instance, by using methods from graph theory;

[154]. However, most early solutions suffered from problems; such as inaccuracy and ambiguities [98]. The most notable algorithm to emerge, in the field of contour extraction, was marching cubes [197].

The marching cubes algorithm moves through the volume and examines one voxel at a time; a voxel is an imaginary cube situated between the volume data points. The goal of the marching cubes algorithm is to examine the voxel-corner intensities and to arrive at a suitable polygonal-mesh to represent the isosurface. By examining whether the corner intensities are above or below the desired isovalue, cases can be formed. Specifically, the state of each corner intensity in relation to the chosen isovalue is converted to a boolean bit in an 8-bit integer; where each bit represents a corner. The naive approach is to consider that there are 2^8 cases and thus 2^8 possible polygonal configurations; although, when considering reflection and rotation, there are only 15-unique cases. There are, however, ambiguities with the original algorithm [86]. Marching cubes was, and still is, infamous because it was patented. As a result of these problems, marching tetrahedra [108, 56] was introduced, where a voxel is dissected into 6 tetrahedrons. This makes the number of cases shrink to 16 and removes any ambiguity, while also bypassing the patent.

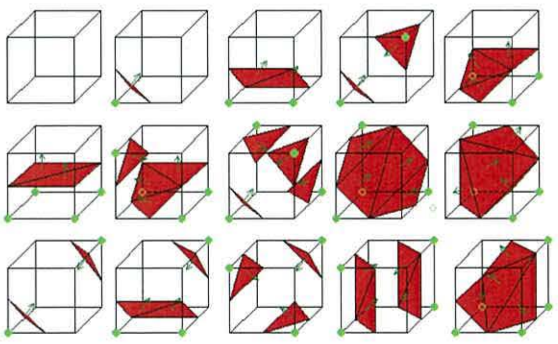


Figure 2.1: The Marching-Cubes voxel cases and resulting polygons. Image source http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html

Employing isosurface extraction and visualizing the resulting polygonal mesh is still popular today. Almost every piece of software that can visualize volumetric data will employ isosurface extraction; for example Matlab [211].

During the early 1990s, focus moved away from how to extract polygonal isosurfaces and moved towards how to find valid regions of the volume quickly. Originally Marching Cubes was designed to march through the entire data-set, which was not ideal for larger data-sets with much empty space.

An important development in speeding up isosurface generation was the space-efficient pointerless octree by [325]. In this work, the volume was subdivided using an octree, with each node containing a minimum and maximum value for the sub-volume the node represented. The research showed that by employing an octree acceleration structure not only could the valid regions of the volume be found very quickly, and thus generate the isosurface much faster than the marching method, but the structure could also be memory efficient.

The Span-Space technique was introduced by Livnat [196] to as another possible solution to the valid-voxel search problem. In a span-space each voxel is mapped to a two-dimensional grid. The dimensions of the grid are the minimum and maximum value stored within a voxel. For memory efficiency the span-space is divided into memory buckets. Data-points (voxels) are then quantized and stored within these buckets. For fast lookup, a kd-tree is also built over the span-space. When searching for given an isovalue, the kd-tree can rapidly find the buckets which contain voxels with the desired isovalue passing through them. The main drawback of the work was the considerable amount of overhead, i.e. voxels checked but not contributing to the final isosurface, due to use of memory buckets.

With researchers solving the search time bottleneck by using acceleration structures focus moved once more toward making the whole process of extraction and visualisation more accurate [220] and faster [196, 64]; of particular interest is the work by Livnat [195] where the isosurface extraction process is accelerated by exploiting view-dependence.

With consumer graphics hardware becoming more widespread, researchers began to exploit them for isosurface visualization [320, 90, 321]. With the work of Rottger [263] an improved Projected Tetrahedra algorithm was presented for isosurface visualization. The work utilized 3D Textures, a multiple pass renderer and exploited OpenGL-hardware boolean operations to speed up the projection and shading of the tetrahedra polygons. Further work with the rendering of tetrahedral cells was presented by Weiler [319], who implemented a ray casting system using a programmable shading language. This removed the need to project the cells using polygons and directly ray traced the tetrahedrons with scene traversal and intersection testing performed on the GPU itself.

2.1.2 Isosurface Raytracing

Unlike surface contouring, which typically extracts the surface to geometry primitives and then renders the geometry with rasterization, ray tracing directly accesses the volume data and immediately renders the isosurface to the screen. The typical method is to shoot rays into the volume and find the first isosurface intersection.

Intersection testing is an important aspect of isosurface raytracing and has various methods available to balance accuracy versus speed [208]. The analytic method [241, 280], employing schwarze's cubic solver [275], is accurate but computationally expensive. Linear interpolation between the entry and exit points of a voxel is the most basic and fastest numerical approach for intersection, but is not accurate. Neubauer's Method [222] uses repeated linear interpolation to arrive at the correct intersection location; however, several iterations are required. Finally, the correct root finding method, introduced by Marmitt, *et al* [208], is able to reproduce the same results as the analytic method in a numeric fashion.

2.1.2.1 History

It was Parker [241] who introduced interactive isosurface ray tracing. The work demonstrated that it was feasible to directly ray trace the isosurface, rather than extract a geometric mesh from the volume. The algorithm employed was a brute-force raytracer with three-steps: traverse the rays through the volume cells, analytically compute the isosurface-ray intersections upon finding a valid cell and, upon a successful intersection, shade the screen pixel. Of interest is the use of bricking and a two-level hierarchy, to help reduce the overhead of empty space and improve memory caching. In addition the use of an analytical-solver for the intersection test provided high accuracy. Finally, the paper itself was popular and served as a generalization of previous work and a base for further work.

During the late 1990s, consumer hardware for computer networking saw vast improvements, allowing for very fast and high-bandwidth communications between machines. Researchers exploited fast and cheap networking to separate the visualization of a data-set across multiple machines [63, 20, 62]. While employing multiple machines was the norm many years previously, these new out-of-core methods provided for real-time visualisation and interactive manipulation.

An important milestone was the introduction of an implicit kd-tree for isosurface rendering on CPUs, by Wald [312]. The work was further improved upon by Grob

[107] to include MIP rendering. The implicit kd-tree was designed to keep a low memory footprint, while also enabling high quality rendering of medium sized volumes, at interactive frame rates on consumer CPUs. The implicit kd-tree method is further explored in Chapter 2.

The rendering of discrete isosurfaces (distance fields) was explored by [118] for massive data-sets. Accelerated by an Octree for space skipping and an OpenGL-shader based raytracing mechanism, interactive rendering with advanced surface-shading was achieved. Specifically, surface curvature was exploited for non-photo realistic rendering, which allowed for highlighting of interesting features on surfaces that would have normally have been hidden. In similar work, Stegmaier [290] presented a ray tracing system for distance fields with more emphasis on ray tracing effects, such as reflection and refraction. Also of note in this work was the ability to combine rendering methods, such as isosurface rendering and direct volume rendering with advanced effects and tone-shading. Finally, work with distance fields also gave rise to methods for converting geometry to volume data, such that volume rendering techniques could be used [142].

A major advancement in progress isosurface came with the interactive isosurface ray tracing of time-varying tetrahedral volumes [311]. This work focused purely on a CPU-based implementation, exploiting SIMD for small-packet and large-packet traversal. For raytracing arbitrary isovalues, an implicit BVH was introduced, much like in the previous work of Wald [312].

With data-set size growing ever larger, research focused on solving the memory problems and rendering speed shortcomings, as was the case with the work by Friedrich [96]. An LOD system was utilized, whereby, upon traversal into a region, the volume data and sub-tree would be loaded into main memory from the hard-drive.

Interactive frame rates were achieved for the isosurface rendering of multi-gigabyte volumes, so large they could not be directly loaded into main memory. In addition, the work did not require Out-of-core methods, unlike the previous research by Chiang [63].

As GPUs improved over time, with far better architecture to provide general programming, the distinction between research into isosurface visualisation and direct volume rendering began to break down. Specifically, it was becoming possible to render isosurfaces with direct volume rendering, simply by choosing a transfer function to highlight the isovalue [90]. Of particular interest was the work of Hadwiger [118] which showed a robust and comprehensive rendering application could accommodate a multitude of rendering techniques and effects, including direct volume rendering and isosurface rendering. However, early isosurface-like transfer functions simply assumed all values above an isovalue should be rendered. Actual isosurface transfer functions require very narrow peaks for a single isosurface, rather than isobands. The main problem with this was discussed and solved by Knoll [162]. The simple solution was to include a peak-finding algorithm into the direct volume rendering integral, such that narrow peaks in the transfer function (isosurfaces) are not missed due to the numerical sampling rate employed by the volume ray tracer.

2.1.3 Direct Volume Rendering

Direct volume rendering reconstructs how light interacts with a gaseous medium and how the light is transported to the viewer's eye. The techniques employed typically can be separated into two fields, photo-realistic rendering and non-photo realistic rendering. For example, a realistic rendering of a volume might be the modelling of radiation through the medium, such as X-Ray, while a non-photo realistic rendering may involve the use of a transfer function to map data values to arbitrary colours and opacities.

The goal in volume rendering is the mapping of information contained in a volumetric medium such that it can be represented on display devices, or printed. Typically, the volume data is stored in a discrete form, as illustrated in Fig.2.2. There are several factors governing the final result, such as: how light propagates through the medium; what the physical properties of the medium are; and what kind of information is being visualized.

2.1.3.1 Modeling Light Transportation

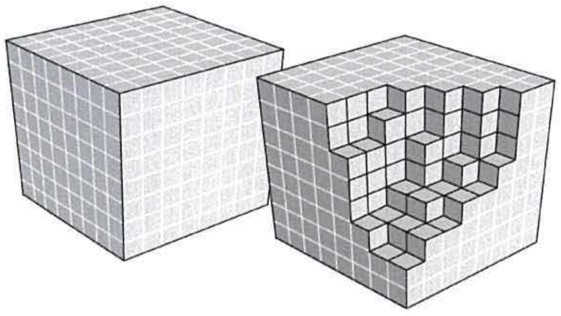


Figure 2.2: Discrete storage of volumetric data and representation as voxels. Image source and copyright: SIGGRAPH 2009 Course Notes [207]

The human eye detects light as it hits the retina. Light, either from the sun or from artificial lighting, must travel from its source through the world before arriving at the eye. As light travels through the world it is altered by the various mediums, whether they are gaseous or solid, that are present. Basic interactions that can occur are absorption, reflection, refraction.

A computer simulation may or may not model all these interactions and indeed may not simulate light as it occurs in the real world. For the sake of speed it is typical to reverse the model such that we only compute the light which enters the eye and not all the light given from a light-source. The most important optical models used today in volume rendering were reported in the survey paper by Max[212] and have been summarized by Engel [89]:

Absorption only: is where the volume medium is assumed to be full of light absorbing gas. No light is emitted.

Emission only: is where the gas within the volume emits light, but is completely transparent.

Absorption plus emission: is a combination of the previous two optical models and is the typically used method. The gas not only emits light, but also absorbs incoming light also.

Scattering and shading/shadowing: is another popular optical method, whereby light is scattered within the volume as it passes through. This method can also consider whether the light between the source and voxel is impeded by the volume, and therefore cast a shadow upon the voxel in question.

For a comprehensive look into the simulation and modelling of light see the two-volume work by Glassner [100].

2.1.3.2 The Volume Rendering Pipeline

There are several stages in a typical volume rendering pipeline [89]:

Data Traversal The data of the volume must be traversed in the sense of acquiring the data necessary to compute the volume rendering integral. Traversal of the data may be accelerated using an acceleration structure; i.e. to avoid empty space [78]. A basic approach is to simply step along rays in discrete steps, accessing the volume along the way. The step-size can also be made adaptive [185].

Interpolation Due the fact volume data is typically in discrete form, it is required to reconstruct the original volume function using filters [210]. The most common filter is simply linear interpolation in three-dimensional space (trilinear). This is especially true for GPU volume renderers as trilinear-interpolation is hardware accelerated.

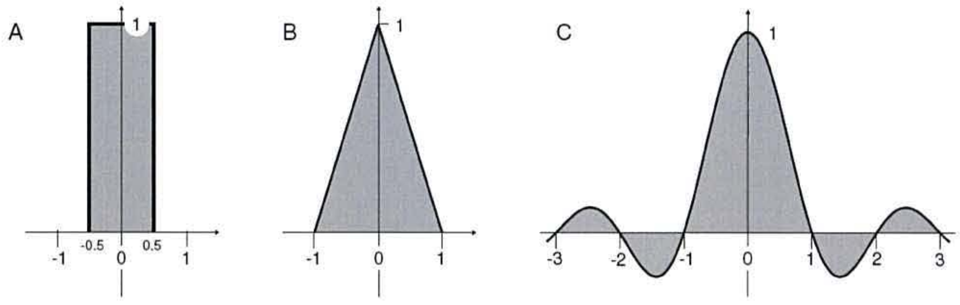


Figure 2.3: Example of reconstruction filters (one-dimensional), where A) is the box-filter, B) is the linear-filter and C) is the cosine-filter. Source: SIGGRAPH 2009 Course Notes [207]

Gradient Computation It is typical to compute the local gradient of data when applying local illumination. Typically, central-differences is used for reconstructing the first-derivative in a numerical manner. It is also common to increase the distance parameter in order to acquire smoother results and limit the visual effect of the discrete nature of the data. Alternative methods may employ the first-derivative of the trilinear-interpolant [241], or tricubic-interpolant [144] or indeed other reconstruction filters where an analytical derivative is available.

Classification Classification, in the sense of the the volume rendering pipeline, is the process of altering the visual properties of specific data or regions within the volume. Typically, this involves applying a transfer function, where the volume data is mapped to a colour and opacity contribution and is in-turn used during composition. Alternatively, a second volume may be utilized that defines a segmentation class for volume regions. Such segmentation classes may represent individual anatomy and as such may be rendered with different optical properties. Pre-classification is the term used when the volume samples are classified prior to interpolation, while post-classification is when the interpolated value is used in the transfer function instead. Pre-integrated classification is when the integral between two values of a transfer function are pre-computed. Pre-integration provides the best rendering quality, while pre-classification the worst [90]. See Section 2.5.

Illumination Local-illumination is the typical lighting method used during volume rendering, which computes the first bound in relation to the scene lights and viewing camera. With additional ray casting, shadowing can be achieved. Global-illumination tracks all light rays, throughout a scene, as they bounce multiple-times until arriving at the camera.

Composition Composition is the process of iteratively stepping along a ray, either in back-to-front or front-to-back ordering, and numerically integrating the volume-integral.

2.1.3.3 Volume Rendering Integral and Composition

Simulating the propagation of light through a medium requires computing the volume rendering integral, i.e., the integration of the medium and optical properties as they travel to the eye.

The typically used optical model is the emission-absorption model and leads to the following volume-rendering integral, as described by Engel [89]:

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)\mathrm{d}t} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)\mathrm{d}t} \mathrm{d}s$$

where κ is the absorption coefficient and q is the source term (emission). Integration is from the entry point $s = s_0$ to the exit location $s = D$. It is common practice to compute the volume rendering integral in an iterative fashion. This leads to the front-to-back (from the camera into the volume) composition scheme, with the assumption that non-*associated colours* are used:

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})\alpha_{src}C_{src}$$

$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src}$$

where C_{dst} and α_{dst} are the accumulated results of the previous computations, and where C_{src} and α_{src} are the source terms, typically given by the transfer function.

2.1.3.4 History

Early research focused on the reprojection of the volume to allow the data to be viewed from arbitrary view angles. Specifically, the additive reprojection technique simulated an x-ray image by averaging the intensities of voxels, situated along parallel rays, from the rotated volume to the image plane [259, 129]. Simular work involved source-attenuation reprojection [139, 273], which assigns a strength and attenuation (opacity) coefficient to each source voxel. This technique allowed for hiding uninteresting areas and to obscure certain objects from view, while highlighting more important features. Another interesting development was depth cueing [304], whereby the volume opacity contribution is inversely-proportional to the distance from the camera to the voxel. This has the effect of giving the perception of depth, which is especially useful when employing parallel projection; i.e., a lack of perspective.

During the early 1980s colour displays became more readily available, which resulted in volume rendering exploiting the benefits of coloured rendering. Specifically, Farrell [92] introduced colour transfer functions by classifying volume intensity ranges to a 9-bit colour (3 bits per colour). This work also reported difficulties in distinguishing the seperate regions when using more than four colours.

An important milestone was the work by Drebin [84], who introduced a basic form of utilizing multiple transfer functions for multiple materials. Specifically, the work analyzed the volume histogram and classified intensity ranges as being one or more materials. For example, very low intensities would be defined as air, while very high intensities would be defined as bone. Rather than illustrate the entire volume with a single colour and opacity, each material was coloured differently and materials could overlap one another. The research is also notable for the composition of multiple input-sources; i.e., the material boundaries, the partial-derivatives for each axis and the individual material volumes.

With desktop computers becoming more readily available to researchers during the late 1980s and early 1990s, there was an explosion of new research. New techniques emerged employing more computationally expensive algorithms while reducing the algorithm complexity. Notable is the work of Levoy [189] which utilized ray tracing and trilinear interpolation in a generalized and simple algorithm; one typically used today. Levoy's [189] approach is further described in section 2.2.1.1.

Optimization of volume rendering became a focus of the work by Levoy [190]. The ray casting method employed was a grid traverser and two methods for performance optimization were described. Firstly an octree-like pyramid of sub-volume cells was built over the original volume space. This structure had the purpose of defining empty-space, such that when the ray traversed into empty cells, the space could be avoided. If the cell was valid, the ray would traverse down the pyramid-tree into a finer volume grid. The technique was further refined in later work by Levoy[191], where a mip-map was employed for gaze-directed rendering. The second optimization employed was early ray termination, which had been researched previously by Whitted[324]. As the ray traverses from front-to-back, colour and opacity is accumulated accordingly. Once the ray accumulation is fully opaque, further traversal is not required (and thus terminated) as no further contribution to the volume integral is possible.

Another major milestone for direct volume rendering was high-quality pre-integration by Engel [90]. Pre-integrated classification, as reported earlier by Rottger [263] for cell-projected tetrahedra rendering of isosurfaces, provides for a substantive improvement to image quality compared to post-classification methods. One of the main drawbacks to direct volume rendering is the need for numerical integration of the volume rendering integral. The problem arises when the sampling rate (along the viewing ray) is too large, resulting in visible artifacts in the final output. Direct volume rendering with pre-integration of the transfer function, forms a lookup into a two-dimensional table, given the volume values at the entry and exit points (i.e. $[t, t + samplerate]$), and uses the value for colour and opacity accumulation. The table itself is populated with the correct integration of all possible value combinations; i.e. the integration of the transfer function from any value, to any value. As a result of using pre-integrated classification, larger sampling distances can be employed. This greatly reduces the render time, without compromising the rendering quality.

The work by Kruger [170] is an example of early work with GPU acceleration of volume rendering. This work was performed on older hardware, where rasterization shaders were employed. The approach had the benefit of being easily integrated with OpenGl or DirectX rendered scenes.

An interesting development was the work by Hadwiger [114] for volume rendering of segmented data sets. The work focused on working with the segmentation data directly, rather than the original volume and facilitated the merger of multiple visualization techniques, such as direct volume rendering and maximum intensity projection. The interesting factor of this work was the trilinear interpolation within the voxels. Due to voxel corners having segmentation IDs rather than intensity values, direct trilinear interpolation is not possible. Instead multiple trilinear interpolations, at least one for each unique segmentation ID within the voxel is required. For each unique segmentation ID, a binary-valued voxel is formed. The interpolated values for each ID case is then stored. The segmentation case with the highest value is chosen and the segmentation ID returned to the shader program. Using this system, high-quality post-classification was possible, as well as pre-integrated classification.

With advancements being made toward more optimal visualization algorithms, as well as further computational power for researchers to exploit, more focus was directed to the visual information portrayed by the rendering; such as multi-dimensional transfer functions as reported by Kniss [159] and the context-preserving rendering approach by Bruckner [37]. These techniques are discussed further in Section 2.5.

The latest research in direct volume rendering suggests that solving the problem of visualizing massive volumes is the current research trend. Of note is the research by Crassin [73], which employs a LOD system, similar to that of Friendrich [96], and load-on-demand bricking. For a broad overview of current techniques the book by Engel [89] is an invaluable survey of real-time rendering methods. In addition the SigGraph course-notes [207] details the cutting-edge research and practical discussions for the implementation of latest visualization methods.

2.2 Rendering Techniques

While ray tracing had always been of interest to researchers, alternate algorithms for volume rendering began to be published in the 1990s, such as those reported in the survey paper by Elvins [88]. These algorithms could be divided into two groups, image-order and object-order. With image-order rendering each pixel is only concerned with the data that the ray of light passes through as it travels from the camera, through the pixel and into the scene. With object-order rendering, individual objects are projected to the screen to determine the pixels that the object contributes to.

The following section outlines the most notable volume rendering techniques to have emerged; these being Ray Tracing, Shear Warp Transform, Texture Slicing, Splatting and Cell Projection.

2.2.1 Ray tracing

Ray tracing projects a ray from an origin (typically the camera) into a scene and can be used for a number of purposes. In relation to graphics, ray tracing typically refers to finding intersections with geometry and/or determining the interaction of the ray of light with a volumetric medium. Ray tracing and ray casting are very similar, except for subtle differences. With ray casting, new directions for the ray are not computed, as they are with ray tracing upon intersecting with geometry. This means effects such as refraction and reflection are only possible with ray tracing. In recent years the term "ray tracing" has been more widely employed to describe both methods.

Ray casting, for computer graphics, was first pioneered by Appel [13] for the rendering of solid geometry, however, it was not until the work of Roth [262] before the term "Ray Casting" was coined. Initial focus was directed toward ray tracing solid geometry and improving the shading model to include more effects, such as shadows [13, 35], specular reflection [315], transparency [223], global illumination [324], and improved realism [18, 29, 31, 151, 328]. With a research foundation for ray tracing solid geometry formed (and later generalized by Kajiya [146]), focus moved toward the ray tracing of volumetric densities and volumetric effects. The work of Blinn [30] focused on improving the lighting models for computer rendering, with specific focus on clouds and how they scatter the light (reflection and diffusion).

An alternative technique to Ray Tracing is Path Tracing, which is a technique used to simulate the physical behaviour of light as closely as possible. Originally described by the rendering equation introduced by Kajiya [146], Path Tracing was later refined by Lafortune [178] to include bidirectional path tracing; i.e. tracing the path of light both from the camera and from the light at the same time. Path Tracing belongs to a group of algorithms able to simulate global illumination (indirect lighting). Other techniques for global illumination include photon mapping [141], radiosity [104], beam tracing [128], cone tracing [5] and ambient occlusion [44].

2.2.1.1 Volume Ray tracing

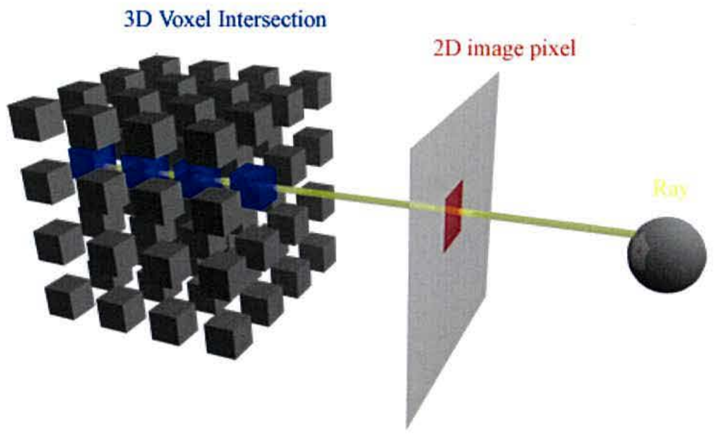


Figure 2.4: Visualization of the Ray casting process. Image source: volviz.com

Work dedicated to ray tracing of volumetric data saw much pioneering research in the mid-to-late 1980s. Of note is the rendering equation described by Kajiya [147], which improved upon Blinn's [30] light scattering model. In addition, the work described a method for ray tracing density models, which, for example, could be used to simulate clouds.

Improvements and generalization of the ray tracing of volume data came with the work by Levoy [189], whose work is notable for the non-binary classification of the data, much like the research published in the same year by Drebin [84]. Levoy's [189] work is also interesting for using a ray stepping algorithm and employing trilinear interpolation within volume voxels. Unlike modern volume ray tracing methods, whereby the classification is typically performed prior to interpolation, Levoy prepared and classified the data prior to rendering. The general pipeline presented first feeds the CT data set into a shading program, which outputs a colour volume. The CT data is then sent to separate classification program, which outputs a volume of opacities. This results in two separate pre-processed volume arrays. The final step of the pipeline casts rays into both volumes and uses trilinear interpolation to sample the data. This process of applying a classification routine before interpolation is referred to as *pre-classification*, whereas interpolating the data before classifying it is referred to as *post-classification*.

Grid traversal was an important method for the earlier research into volume ray tracing. Typically there are two methods for tracing a ray through a volume. First we can progress along the ray at constant distances (sampling rate), which requires floating point arithmetic to compute the new location within the volume. The alternative is to traverse the ray using a DDA line algorithm, which is useful when only integer arithmetic is feasible. Initial use of DDA ray traversal focused on traversing polygon scenes that had been space-partitioned into grids, as detailed by Fujimoto [99] and Amanatides [6]. Later these techniques were used for interactive ray tracing of volume data [190]. In modern times, grid traversal methods showed favourable results for the rendering of animated scenes, as well as providing efficiency due to ray coherence, in work by Wald [313].

Exploration of volume data and providing multiple visualization options, such as surface clipping, was the focus of the work by Hohne [130]. A generalized voxel model was introduced by this work, which enabled the introduction of a second imaging modality; i.e. combining CT and MRI data. Ray casting was utilized for the rendering of segmented surfaces from the volume. The work also computed the surface normal from the original volume data using central differences, which produced superior image quality.

Ray tracing, typically being a naturally parallel process, exhibits coherence among primary rays. This is to say that rays traversing through the same areas of the scene can exploit caching, such that data need only be loaded once for a group (or packet) of rays. Initial research into exploiting ray coherence came with the work by Kaplan [149]. It was further explored by Yagel [332], who solved the problem of using DDA algorithms without artefacts. Later, other researchers would exploit coherence for other visualization methods, such as Wilhelms [326], who outlined a coherent method for cell projection.

Ray coherence was again exploited by Wald [314] to enable interactive ray tracing of large polygon scenes and later introduced for animated scenes [313]. The work is also notable for introducing ray packets, which could exploit SIMD technology for considerable performance gains. In general, coherence algorithms typically have used a DDA grid traversal, with packets of rays. The latest research by Knoll [163] continues this trend using a multi-resolution grid for the ray tracing of isosurfaces at interactive frame-rates on CPUs.

As consumer hardware technology advanced, the speed with which volume rendering could be achieved was significantly improved as reported by Westermann [321].

2.2.1.2 Volume Ray tracing on GPUs

With the advent of programmable shader languages, ray tracing pipelines could be implemented purely in the GPU, as was presented by Purcell [252, 253]. Alternatively, the GPU could be exploited only for the computationally intensive aspects of the ray tracing pipeline to create a hybrid CPU/GPU rendering system [52]. However, while the power of GPUs is undeniable for certain functions (typically those of high computation and low thread-branching), an on-going battle has been waged between CPU-based ray tracers and GPU-based ray tracers, as to which is the fastest and best.

Ray tracing on GPUs introduces new challenges for researchers (as reported by Aila [4]), not only from an implementation perspective but also in determining which algorithm can best fully utilizes the device hardware. Many previous ray tracing approaches were designed for CPU-based ray tracing and as such may not have the same benefits when implemented on GPUs. Ray tracing algorithms can be divided into two categories; those that reduce the overall workload and those that optimize the traversal. For instance packet traversal [312] is where a group of rays are represented and traversed as a packet. Packet traversal reduces the amount of work for the group of rays by performing the common traversal steps once. Coherent traversal [164] attempts to exploit the fact that rays typically traverse the same nodes most of the time by forcing convergence after ray divergence. Optimization algorithms cater for specific strengths and drawbacks of targeted architecture, for example, stackless traversal [94, 248, 73, 133].

A major contribution to GPU-based volume ray casting was the stream framework proposed by Kruger [171]. While the framework utilized standard acceleration techniques, such as empty-space skipping and early ray termination, the method by which the rendering method integrated into the GPU was unique. The proposed method renders a cube, which has its RBG colours set to its XYZ values. The initial rendering pass only renders the back-facing polygons of the cube. The resulting image is stored in a GPU texture. Next a ray tracing shader is attached to the GPU and the cube is rendered again, except this time showing the front faces. Both rendered images (front and back faces) provide the start and end positions for the rays. Each front-face fragment (pixel) then becomes a ray and is shaded by the GPU using the attached ray tracing shader. The shader simply subtracts the RGB values of the front face from the back face, which results in a view direction vector. The ray tracer then steps along the ray, samples the volume, which is stored in a 3D-Texture, and computes the volume integral.

The multi-pass rendering framework by Kruger [171] proved useful for researchers, with Hadwiger [118] extending the idea for larger volumes. In Hadwiger's [118] work the volume was divided into bricks, typically 8^3 in size. Rather than render a cube, for computing the start and end points to ray trace, each valid brick was rendered. This enabled a far better system for empty space skipping than the previous work. In addition the ray tracer did not sample the original volume, but rather a compacted volume of bricks. This ensured that no empty-space was uploaded to the GPU, thus saving time and memory space. Also of note is that bricks could be uploaded asynchronously, such that the ray tracer could render one brick (and read the data) while another brick was being uploaded. This facilitated the rendering of large volumes without the data transfer becoming a bottleneck.

An alternative framework for GPU-based volume rendering, by Stegmaier [290], was similar to Hadwiger's [171] work in that the front-faces of a cube were rendered with a custom fragment-shader attached to enable ray tracing. However, the later work used a single-pass and omitted the rendering of the back buffer. Ray directions were computed on-the-fly by subtracting the polygon-fragment location from the camera location. This work is also notable for implementing a wide variety of advanced effects, such as shadows, reflection and refraction.

2.2.1.3 Stackless Ray tracing

In general, ray tracing of an acceleration structure requires a *stack* to record tree nodes which cannot be immediately traversed, but which should be returned to if required. Due to shared-memory limitations of GPUs, such an approach typically requires the stack to be stored in the GPU's main memory (global memory). However, since the slowest aspect of most current GPUs is accessing the global memory, a stack-based approach can induce a memory bottleneck.

Foley [94] highlighted that a stack-based traversal approach on GPUs induced such a performance bottleneck. Their solution was to completely remove the need for a stack and resort to extra computation for correct traversal. The two approaches they introduced for general kd-trees were kd-restart and kd-backtrack. kd-restart, upon requiring to return, restarted traversal from the root node, while also moving the ray's segment-range ahead of previously visited areas. kd-backtrack dealt with returning by backtracking up the tree one node at a time until a valid continuation point was found. Both approaches require a considerable amount of additional work to find valid nodes into which to continue traversal.

Further exploration of stackless traversal was the focus of the work by Popov [248] with the rediscovery of Ropes [270, 271, 202]. The Ropes technique adds additional neighbour information to each node of a tree, which allows a traversal mechanism to traverse into a neighbour cell/region upon discovering that the current sub-tree has no ray/geometry intersection. Unlike a stack though, intermediate nodes above the neighbour node can also be avoided, resulting in less nodes being visited in general; subsequently making the method faster than a stack anyway. However, the massive memory cost of requiring pointers for each plane of the bound is a disadvantage.

Stackless approaches are typically a game of balance, such that by avoiding a stack either more memory is required [248] in the tree, or more computation and iterations of the traversal mechanism is needed [94]. The work of Horn [133] focused on the balance aspect of designing a stackless method by introducing kd-shortstack. Rather than completely remove the stack, a very small one is used. This short stack can be made small enough to fit in the fast (but limited) shared-memory available on the GPU multiprocessors. If the stack overflows the slower global memory can be used instead [200], or the algorithm can revert to kd-restart [133, 94].

2.2.2 Texture Slicing

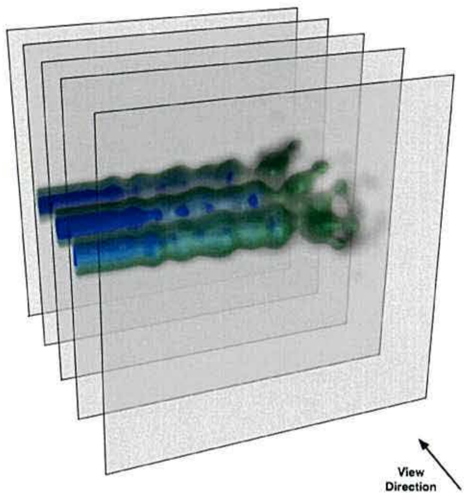


Figure 2.5: Example of Texture Slicing. Image source: equalizergraphics.com

Texture slicing, depicted in Fig. 2.5 is a very simple, yet effective approach for visualizing volumetric data. Several two-dimensional slices are placed within the volume, facing the observer, at regular intervals. The underlying volume is then sampled on the slices. These slices are then composed together by the graphics card to form the final image. This approach is especially useful for graphics hardware as all sampling and composition can be accelerated by purpose-made hardware components.

The use of 3D texture mapping for volume rendering was explored simultaneously by Cullip [74] and Crabal [48], implemented on the RealityEngine workstation with OpenGL extensions. Two variants were discussed, one where planes were parallel to the observer and another with planes parallel to one of the volume's axis. Later work by Rezk [257] enhanced the slicing technique to overcome accuracy problems when using 2D Texture mapping on consumer hardware where 3D Texture mapping was not available. An alternative to placing 2D planes within the volume's Cartesian domain was the spherical-domain based approach of Westermann [320].

2.2.3 Cell Projection

Like splatting, cell projection is an object-order approach. However, unlike splatting, the cell is typically polygonized and the polygons projected to the screen. Early work by Shirley[282] decomposed volumes into tetrahedral cells. These were then polygonized and rendered (typically with hardware graphics cards) with transparent triangles. The rendered triangles were composed to form the final volumetric rendering. In similar work, Wilhelms [326] projected the visible cell faces directly (rather than using tetrahedral calls), using triangles or quadrilaterals, and improve image quality with improved volume integration methods. Renewed interest in cell projection by Rottger [263] exploited new graphics hardware to further improve the rendering quality and speed for direct volume rendering and isosurfacing.

2.2.4 Splatting

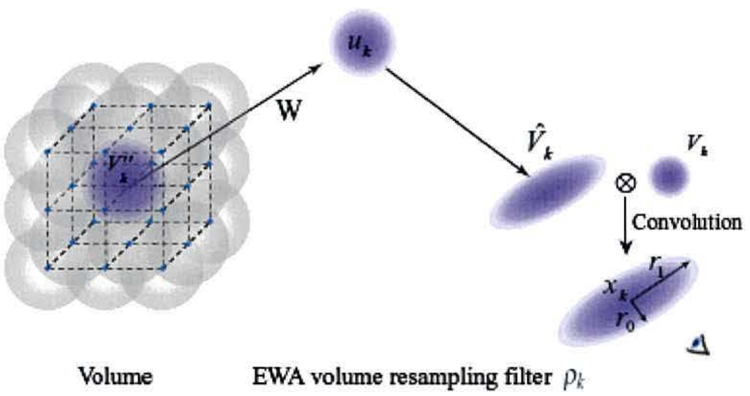


Figure 2.6: Example of the Volume Splatting Process. Image source: [61]

Splatting is a method whereby the volume, in the form of three-dimensional reconstruction kernels or splatting *primitives*, is projected to the image plane; see Fig. 2.7. These projected kernels are first sorted, then evaluated and finally composed to form the final image. The method is very flexible and only requires some form of sorting to ensure correct composition ordering is met. The first publication for volume splatting was by Westover [322, 323].

A hierarchical approach was presented by Laur [182], where a pyramid structure was used to refine the splat cell size based on local data variance. Also the pyramid structure allowed rendering to be flexible so that lower resolution could be chosen to meet a desired frame-rate.

A new splatting primitive, called the EWA (elliptical weighted average) volume resampling filter, was introduced by Zwicker [343]. This filter combined an elliptical Gaussian reconstruction kernel with a Gaussian low-pass filter to deliver reconstruction of both surface (isosurfacing) and volume data for perspective-based visualization with improved anti-aliasing. The work was extended by Ren [256] with a new object-space formulation of EWA splatting for irregular point samples and an efficient implementation on graphics hardware. Hardware acceleration was again the focus of Chen [61], who introduced an adaptive EWA splatting method without reduction in quality, and Neophytou [221] who introduced RBF-based (Radial Basis Function) splatting primitives for irregular grids.

2.2.5 Shear-warp

Shear-warp volume rendering, introduced by Lacroute [176] is closely related to texture slicing in that two-dimensional slices are placed within the volume, however, these slices are axis-aligned rather than aligned toward the camera. This gives the benefit of the slice-sampling being completely linear and subsequently very optimized for memory access. The approach does, however, require an additional step to correct the visu-

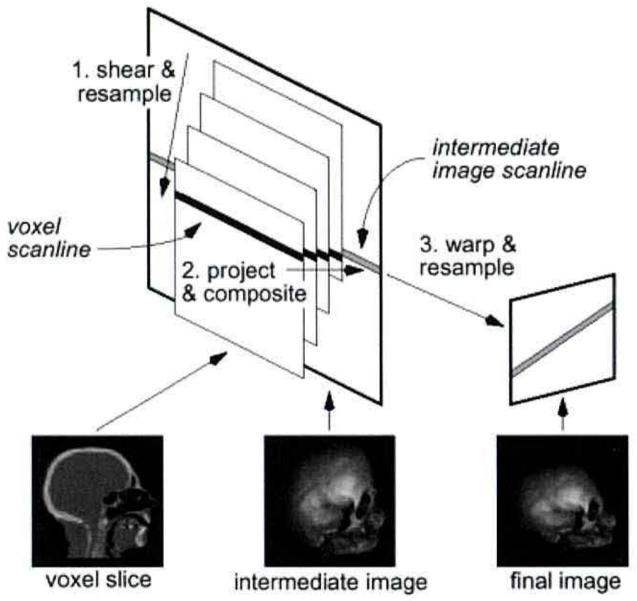


Figure 2.7: The Shear-warp volume rendering process. Image source: [176]

alization for proper composition. Specifically, once the volume slices have been rendered to an intermediate base-plane, the base-plane must be transformed using shearing to match the view perspective. While the method allows for much optimized memory access, the shear-warp transformation means certain regions of the image-plane may be under-sampled (poor quality). As such, it is typical to super-sample during the visualization so that the base-plane is larger than the output screen. This then allows for better quality after the shear-warp transform. The shear-warp method is depicted in Fig. 2.7

A number of advancements were published in the years after the original work. Utilization of shared-memory multiprocessors by Lacroute [175] facilitated real-time rendering. In independent work, Amin [9] also presented a parallel implementation of the shear-warp algorithm for faster results. Later work by Sweeney [294], shortcomings with image quality were addressed using body-centered cubic grids, while other research by Schulze [274] combined the shear-warp method with pre-integrated classification. The effectiveness of the algorithm was also highlighted for the visualization of 4D (time varying) datasets in the work by Anagnostou [10]. Finally, the technique saw renewed interest for modern graphics hardware with the work by Guo [111]. As modern GPUs are most optimized when global memory is accessed linearly, shear-warp was shown to exploit the hardware texture caching quite well.

2.3 Acceleration Structures

As rendering techniques became more advanced and more complicated scenes could be rendered, researchers began to highlight the need for efficient acceleration structures both for geometry ray tracing [152, 17, 265] and volumetric visualization [190].

Acceleration structures are now an important aspect of modern volume rendering approaches, especially so if large regions of the volume are unimportant or empty. In most rendering approaches the empty space within a volume never contributes to the volume integral. If the empty-space is avoided completely, and assuming the mechanism for avoiding the empty space is faster than examining it in the first place, faster rendering can be achieved.

There are many acceleration structures available and each is typically designed for specific purposes or to have specific properties; be it faster lookup, or a low memory footprint. Object-space and domain-space splitting is also a defining attribute of acceleration structures.

2.3.1 Grid

The most basic acceleration structure available is the sub-division of a volume (or space) into a regular grid. Early ray tracers, such as those by Kajiya [147] and Levoy [190], traversed grids for volume rendering, while Amanatides [6] and Fujimoto [99] also used them for polygon ray tracing.

The work of Parker [241] introduced a multi-level hierarchy grid, where the volume is divided into equally sized cells. These cells were referred to as macrocells as they also contained a min/max value to describe the range of volume values contained within it. Also utilized in the work was bricking, which had been introduced by Cox [72].

Bricking is typically performed as a solution to when the volume is too large to be completely stored in memory, as reported by Westermann [320]. Rather than render the entire volume, each brick is rendered individually. While one brick is being rendered, others can be prepared and loaded, i.e., from the hard-drive to memory. Modern ray tracers have exploited grid traversal for ray coherence resulting in better performance due to optimized memory access upon groups of rays [163, 314].

Gobbetti [101] presented a method for out-of-core volume rendering of massive data-sets. By decomposing the volume into small bricks, asynchronously transferring data to the GPU, and the removal of empty space, the work allowed for real-time visualization of very large volumes at interactive frame rates.

2.3.2 Kd-Tree

A kd-tree, introduced by Bentley [22], is a spatial splitting structure. Given a root node, which encompasses the entire volume, the space is split by an axis-aligned plane. The splitting typically occurs along the axis corresponding to the largest dimension. By recursively splitting nodes, and forming two children, a hierarchical tree is created. Examples of kd-trees being used for visualisation, include storing sampled points during image reconstruction [235], dividing a volume with a balanced kd-tree for isosurface and direct volume ray tracing [312] and for ray tracing of polygonal scenes [95]. Further work has focused on improving the build quality of kd-tree to optimize for ray traversal [102, 202, 123, 137] and to facilitate building (or updating) in real-time [340, 247, 279, 137] on GPUs.

2.3.3 Octree

An octree is another favourite acceleration structure researchers have used for accelerating visualization, such as the work by Meagher [217, 218] and Samet [272]. Given a root node, which represents the entire volume, the space is separated into eight octant children. With an octree, nodes are split along all the axes and when the space can no longer be divided a *leaf* is formed. Because all three of the axes are split at a node, the depth of octrees is typically quite shallow, when compared to a kd-tree. A survey of Octree methods was reported by Knoll [160].

Octrees have been utilized by many researchers for various visualization problems, such as isosurface visualisation [325], representation of three-dimensional objects [138], simplifying object neighbour finding for collision detection [219], hidden surface removal during volume rendering [83, 316], and overcoming volume memory requirements using octree-based compression [180, 112]. Knoll [166] used a lossless-compression octree representation to store compressed volume data for fast iso-surfacing, while Hadwiger [117] described a two-level hierarchical representation utilizing a form of octree, which allowed object-order and image-order empty space skipping for real-time ray-casting of discrete isosurfaces.

An interesting development was the branch-on-need octree, where the nodes were only further split if needed, i.e. if that region was visible and needed to be rendered. Originally introduced by Wilhelms [325] the method was further refined by Sutton [292] for temporal data-sets.

2.3.4 Bounding Volume Hierarchy

Bounding Volume Hierarchies (BVH) have been quite popular for many years (used in the work by Kay [152]) due to their simplicity and more favourable properties than Kd-trees or Octrees. Most notable is the fact that a BVH is an object-order hierarchy rather than a space-order hierarchy. This is to say that a BVH splits a node by the data it represents, rather than the space. Because of the absence of splitting planes, nodes store a boundary, which encompasses the space within which the objects reside. In addition, the traversal of a BVH requires rays to check for intersections with the bound of each node. BVH can be used for volumetric data, however, they are typically better suited for irregular grids, tetrahedral volumes [311] or polygonal data [110, 109].

The basic structure has seen little change over the years, except for two notable cases. Firstly, instead of having three bounds it is possible to only store the bounds for a single axis in each node to form a bounding interval tree, as shown by Wchter [317]; these forms of trees are also referred to as range-trees [23]. Secondly, the work by Dammertz [77] compacted the BVH tree, such that a node was split into four children rather than the typical two, which generally reduced the number of traversal iterations during ray tracing allowing for better performance. Real-time construction of BVHs has also been a focus for researchers, either exploring refitting of existing BVHs [181, 310, 334] or completely building a new BVH from scratch [308, 183]

2.4 Graphics Hardware Programming

Over the years computers saw the introduction of dedicated hardware to accelerate 2D rasterization and then, in later years, 3D graphics acceleration. Initially, hardware was expensive and limited to specialized research workstations. However, in modern times consumer-based hardware that can operate in desktop computers has becoming the *de facto* standard for performing wide-impact research upon [233].

2.4.1 Early Hardware Programming

In order to allow developers (primarily game developers) the option for more control over how 3D geometry was rendered, shader languages were introduced [261, 32]. These shader programs worked primarily on the input geometry and the screen fragments (pixels). Early development of hardware-based visualization methods utilized these shaders in order to accelerate the volume rendering process [171, 114, 192, 140, 278]. However, as shader languages were not designed for general usage programming, much work was dedicated to getting the programs to work in the first place. This typically entailed packing general variables into textures (within the red, green, blue and alpha components). In addition, loops were not available with earlier shader languages.

2.4.2 Compute Unified Device Architecture (CUDA)

Recently a new GPU programming language has been developed called CUDA [225]. CUDA is designed to facilitate general usage programming on GPUs, while also providing the mechanisms for parallel processing. While other languages are in development and have been explored for use with visualization, notably OpenCL [42] and Brook [43], CUDA still remains the most developed and has been shown to be efficient for interactive ray tracing [200]. Parallelization of algorithms has been a difficult endeavor, and while automatic methods have been proposed [342], it typically requires extensive research and a great deal of time.

CUDA allows for code, in the form of kernels, to be uploaded to the GPU and to operate directly on the hardware. In addition, CUDA also allows for simple integration with existing C/C++ code. What is interesting about this integration is that while shader languages may require a great deal of initialization code before work can even begin, most of the low-level operations required to initiate CUDA kernels are hidden. In fact, apart from a minor difference in the calling convention, CUDA kernels can be called much like a normal C function.

CUDA has had a far reaching impact on research in many fields, not just visualization. For example with N-Body simulation [226], SQL Database acceleration [21] and fast detection of humans in videos [26].

2.5 Classification and Context

Volume data acquired from CT or MRI are typically recorded as single value intensities. As colour displays became more readily available researchers began to exploit the benefits for coloured volume rendering. By classifying the input data, it became possible to render different intensities with different colours and opacities. This section details the work with classification and transfer functions. Additionally, research into the segmentation of volume data is also explored. Volume segmentation is the process of defining and separating one or more regions within the volume. Different to classification, which is typically performed during the rendering cycle, segmentation also considers anatomical features, locality, *a priori* knowledge and directed user input.

2.5.1 Transfer Functions

In standard direct volume rendering, visual distinction of objects is usually achieved with a transfer function that assigns optical properties such as colours and opacities to data values [84, 302]. During composition of the volume rendering integral, these colours and opacities are accumulated as the ray passes through the volume and finally rendered to the screen pixels.

Developing methods for the design of transfer functions is a difficult task, as reported by Pfister [244]. For certain image modalities, data intensities can have an intrinsic meaning, such as bone being high-valued, while air being low valued; in such a case, simple transfer function can be easily achieved [84]. However, soft tissue typically has a uniform set of values (for example in CT data) which makes visual distinction hard to achieve.

Marks [206] presented an automatic systems where a variety of transfer functions are automatically generated with various colours and opacities given to various ranges of data. The resulting volume renderings for each automatically generated transfer function are previewed to the user who is then able to choose which to use. Kindlmann [155] presented a novel technique for semi-automatic generation of opacity functions to visualize boundary relations between materials of near-constant value. The work explored designing transfer function over 2D and 3D histograms based upon the scalar value and the first and second derivatives. Kniss [158, 159] expanded upon the work with multidimensional transfer functions and GUI improvements to aid users to manipulate multidimensional transfer functions.

2.5.2 Context/Importance-Driven Rendering

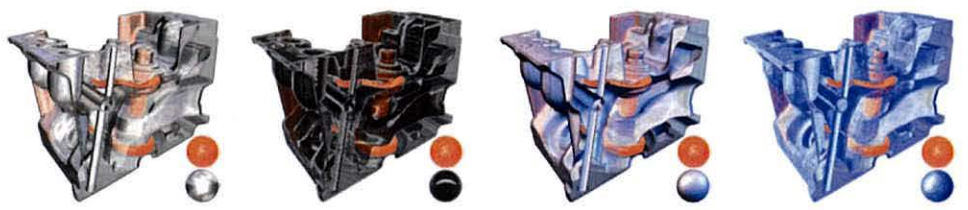


Figure 2.8: Transfer Functions allow for varied visualisations of volumetric data. Image source: Bruckner [40]

Given a large range of possible settings, constructing an appropriate transfer function is often a daunting and frustrating task that involves adjusting a lot of non-intuitive parameters. A solution to this problem is context-preserving rendering, which is where the opacity of samples is modulated to reflect its perception importance to the viewer.

Context-preserving volume rendering, as introduced by Bruckner [37], utilizes the intrinsic information of a volume, such as gradient, and view dependent information, such as camera location, to provide context cues for the viewer. By employing context-preservation the user is typically able to perceive the make-up of many more anatomical objects when compared to standard DVR.

Context-preserving rendering has an artistic background. Many illustrative and technical drawings of medical organs use an artistic method called 'ghosting' to hide surfaces which obscure the details behind them; for example a drawing of the palm of a hand may be made transparent to allow the viewer to see inside. Indeed many researchers have attempted to mimic illustrative techniques [87]. Bruckner's [37] research on context-preserving rendering is able to automatically simulate the 'ghost' effect. However, their approach does not use segmentation data and must simulate the transparency effect outlined by Diepstraten [81]. Similar work by Viola [307] utilized segmentation data in an importance-driven rendering technique to compose images based on the importance of user selected objects; i.e. to create a transparency window on the skin to reveal anatomy within.

2.6 Segmentation

Segmentation is essentially the act of differentiating a specific feature from the remaining features and space. Although segmentation has *some* roots in early volume and image visualization, i.e., classification, it has now become an enormous research field of its own [97, 266, 236, 337, 66, 203] and has shown importance in preoperative planning [36]. A simple example would be the visualization of a human skull as recorded and stored by a CT scan. This would essentially entail removing the flesh and only leaving the bone structure. There are many ways that segmentation can be accomplished.

2.6.1 Overview

The task of segmenting a volume data-set can be achieved manually by hand, automatically, where the user does not initially direct the segmentation, or semi-automatically which requires initialization and/or direction from the user.

2.6.1.1 Manual Segmentation

Manual segmentation, i.e. performed by hand, of scientific data can still be a commonly exercised task [122, 174, 58]. Manual-segmentation could be manually drawing on images, manual thresholding or the outlining of shapes or tissue [153, 67]. Typically, manual segmentation is performed by experts who have extensive knowledge of the data recorded. As such there is an element of trust that the results of an expert's segmentation will be correct. Even for two-dimensional data, such as slides, this can be a difficult task to perform especially if there are several features to differentiate. For three-dimensional and four-dimensional data, the task of manually segmenting each slice would be an extremely long endeavour, even at low resolutions [130]. Not only would manual segmentation be time consuming, it could be subject to errors.

2.6.1.2 Automatic Segmentation

Automatic segmentation is typically utilized where *a priori* knowledge of an object leads to an automatic method to segment that object or feature from any data-set. For example, for the automatic segmentation of the brain [19, 69, 41], lungs [135, 15, 16], heart [338, 243] and cancer tumors [251, 14].

Automatic segmentation techniques, which do not require previous knowledge of the objects present in the data-set, are also possible. These methods typically segment the entire space and result in the separation of all unique objects present in a data-set. Examples include the contour-tree method [303, 269, 50], which typically also make use of thresholding or level-set methods [231] with random seed locations [143].

Kuhnigk [173] presented an automatic, real-time segmentation approach for lesions. In addition, the approach presented a method to analyze the resulting segmentation for accuracy, called segmentation-based partial volume analysis (SPVA).

2.6.1.3 Semi-Automatic Segmentation

Semi-automatic segmentation schemes are methods whereby information is given by a user [229] to the segmentation algorithm in order to direct it toward a custom segmentation [177, 49]. For example, a user may define seed points for a level-set algorithm, or region-growing algorithm [246]. By altering the seed locations, or adding multiple source, different segmentations could be possible. Other examples of semi-automatic segmentation algorithms are machine-learning methods. Machine-learning methods typically 'learn' a small number of training examples. Once trained these machines, when given an arbitrary input, attempt to predict whether the input is part of the desired segmentation.

2.6.2 Region Growing

Region growing is a simple technique where upon being given a seed (starting location) the algorithm will expand the segmentation outward if the neighbouring pixels/voxels have a similar property. Further interaction is possible by the user, by defining constraints such as edges the region cannot expend into. Effective segmentation can be achieved by this popular approach [341, 2, 242, 298, 131].

2.6.3 Watershed

The watershed algorithm considers the volume or image to be a topographic relief. A simple analogy is to consider a drop of water as it falls into a surface. The drop will flow down a path reach a local minimum, or water-basin. Further raindrops will expand the water-basins until regions overflow and merge with one another. The approach has been found to be useful for segmentation of volumetric data [283, 91, 27]. Automatic segmentation using an iterative watershed has also been reported by Mancas [205].

2.6.4 Level-Set Methods

The Level-Set method [232, 276] is a technique used to define shapes, curves and boundaries from functions. An example of how a simple shape is formed in this fashion is shown in Fig. 2.9. Segmentation is achieved by the evolution of the level-set function itself, rather than of the contours, i.e. like region-growing. By using a signed-distance function, inside a shape, the zero-level set will form a boundary. Numerous research has utilized level-set methods for the segmentation of anatomy [238], with specific interest in directing the level-set by using shape *priors*, as reported by Rousson [264] and Chen [60].

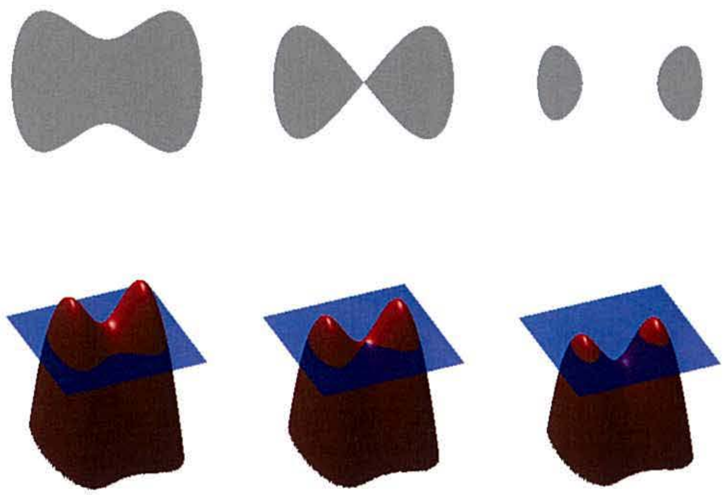


Figure 2.9: Example of the level-set method. Image source: Wikipedia [230]

2.6.5 Deformable Models / Atlas

Segmentation using deformable models involves using 3D shapes with forces acting upon them, which deform the surface towards a shape. These forces typically attempt to constrain the original model, or to contract/expand it toward detected edges or features. The approach is popular for the segmentation of anatomy in medical images [68, 216, 215].

A similar method is Atlas, where an atlas-image is chosen, i.e. an image of the brain, and the system attempts to register that shape in the subject-image. This is achieved by transformation of the Atlas image. While simple, the method has been shown to be very useful [121, 204, 198].

2.6.6 Neural Networks

Neural networks [124] are used to learn patterns [85] based on training-sets, either supervised or unsupervised, and then predict the probability [288] that further input-data matches the trained model. Applications for image and volume segmentation have much interest and success [3, 120, 234, 237].

2.6.7 Clustering

Clustering methods are a broad selection of techniques which subdivide a data-set, where data elements in clusters are expected to be similar and the union of clusters results in the original whole data-set. Hard clustering divides the data into a single cluster only, while fuzzy clustering [25] assigns a membership value of each data-point for each cluster. Clustering methods have been used for image and volume segmentation, for example, Clark [65] segmented MRI volumes while Zhang [336] presented an application of a fuzzy c-means method for segmenting brain grey-matter tissue in MRI images. Shen [277] addressed the noisy nature of results from MRI's and developed a fussy c-means clustering algorithm. The proposed approach focused on neighbour attraction, which considered the location and features of neighbouring voxels. Subsequently, the work showed more accurate segmentation results, of brain tissue from MRI data.

Chapter 3

Previous Work

This chapter details and compares the major research that is relevant to the work undertaken in this thesis.

3.1 Stackless Raytracing

Numerous ray tracers are based on using either a kd-tree, such as the interactive Kd-Tree ray tracers by Horn [133] and Shevtsov [279], or a BVH, such as the BVH raytracer for deformable scenes by Wald [310]. Both Kd-Trees and BVHs are a form of binary-tree, such that nodes partition their represented space or object into two groups. In the case of a kd-tree the space is split using a split-plane, as shown in Fig. 3.1.

Ray tracing algorithms that employ a kd-tree, traverse through the acceleration structure to find geometry, or voxels, likely to intersect the ray. If a ray intersects the splitting plane node, then both children are valid and possibly need to be traversed into. BVHs are much the same, except splitting planes are not used and computing which is the nearest node is done differently.

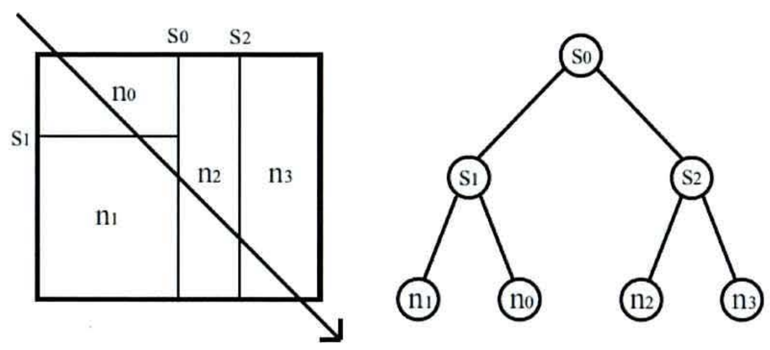


Figure 3.1: Example of a Kd-Tree. Image source: Foley [94]

With a depth-first traversal mechanism, where the algorithm tries to find the first intersection along the ray, the closest node is traversed into first. If the traversal of the first node does not result in the ray termination, then the second node will have to be traversed into. Maintaining the list of *other nodes* to return to is typically achieved with a stack. However, a stack implementation on GPUs typically requires using the slow global-memory. Ultimately, accessing this memory repeatedly results in a slowdown of the ray tracer.

Problems with employing a stack have been addressed by exploring semi-stackless or completely-stackless approaches, such as those described below.

3.1.1 Kd-Restart and Kd-Backtrack

The initial work that focused on stackless ray tracing, on modern GPUs, was undertaken by Foley [94]. In their work two methods were devised for ray tracing general kd-trees without a stack; Kd-Restart and Kd-Backtrack. Both methods where based on the principle of moving the valid ray segment ahead of previously visited nodes.

During ray tracing of kd-trees, a ray segment $\{t_near, t_far\}$ is maintained. This segment is used to determine whether a ray intersects a split plane, or whether the intersection of the ray and plane at t_d occurs before ($t_d < t_near$) or after ($t_d > t_far$) the ray segment. When a return is needed by the traversal mechanism, Kd-Restart sets $t_near = t_far$ so as to avoid the current node. The mechanism then returns to the root node of the tree and resumes traversal. The effect is that the traversal mechanism will travel down the same path as until it reaches the parent of the node just returned from. As the ray segment now lies completely on other side of the split plane, the previously visited node will not be valid and only the alternative node will be traversed to.

Kd-Backtrack works in much the same way as Kd-Restart, except rather than restart from the root node, the traversal mechanism (after progressing the ray segment forward, returns to the immediate parent of the current node. Kd-Backtrack was reported by Foley [94] as being faster than Kd-Restart, however, both approaches will result in previously-visited nodes being tested at least one more time.

Refinement of kd-restart by Horn [133], by moving the root node (the point where traversal restarts from) deeper if possible, reduced the number of extraneous nodes revisited because of traversal restarts. This approach also improved the performance over the original Kd-Restart in the work by Crassin [73], who employ the bricking method for volume rendering of massive data-sets. The work is also of additional interest for partially using indices for traversing their acceleration structure; the stackless approach presented in Chapter 3 is based upon using index numbers, rather than pointers, to traverse a kd-tree.

3.1.2 Kd-Shortstack

Kd-Shortstack, or simply *shortstack*, was a method introduced by Horn [133] that used a small stack rather than a full one. This stack was a fixed-size list, stored in the fast shared-memory onboard the GPU processors; this effectively bypasses the use of the slow global-memory. To compensate for stack overflows the use of Kd-Restart can be used. Specifically, when new nodes are pushed into an already full stack, the bottom most entry is discarded. This has the effect of storing the newest return-points, but forgetting the older ones. To compensate for the lost data, when popping from an empty stack the traversal mechanism will revert to Kd-Restart. The shortstack method was reported by Horn [133] as being more than twice as fast when compared to the original Kd-Restart by Foley [94].

3.1.3 Ropes

An alternative and novel approach to stackless traversal was the re-introduction of *ropes* by Popov [248] for GPU based ray tracing. Ropes are additional memory pointers that link a node to its neighbour nodes. This allows the traversal, when needing to return the next valid node, to simply travel along the rope into the neighbour node. The approach comes at the cost of additional memory pointers per node as well as additional computation to build and optimize the ropes prior to rendering.

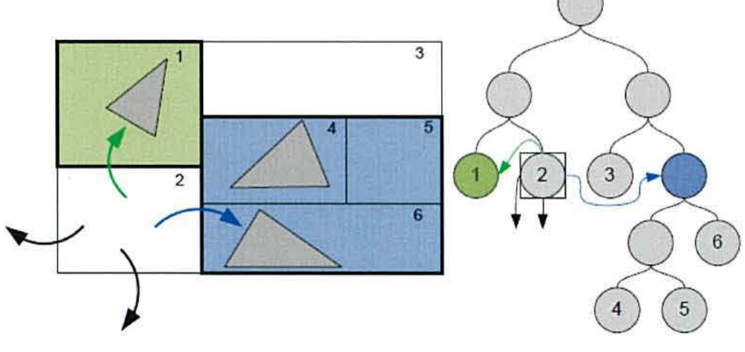


Figure 3.2: Example of Ropes for a Kd-Tree. Nodes are linked to neighbouring nodes via memory pointers (Ropes), once traversal required testing other parts of the tree, the mechanism follows the link for the face the ray intersects. Image source: Popov [248]

The traversal mechanism my Popov [248] is depth first until a sub-branch becomes invalid, at which point the traversal is much like grid-traversal. Each node has 6 faces.. The first task is to determine which face the ray exits through and once determined, the node link of that face, linking the node to its neighbour, is followed. This give an advantage over normal kd-tree traversal in that less intermediate nodes are visited in comparison. Thus, not only is the Ropes technique stackless, it will also have less traversal iterations; both of these factors for GPU-base ray tracing.

The main disadvantage of this method is that a pre-processing step is required to build the face-links before they can be used. This may main that real time building and optimization of the links is not feasible, say for large animated scenes. The other disadvantage is the massive memory requirement; un-optimized each node will require 6 32-bit memory pointers. For example, Popov reported, that on average, the memory overhead was a factor of 3.

3.1.4 Link-Map

The *Link-Map* method [284] is a stackless traversal mechanism for BVH trees. In essence this method stores the traversal order, top-to-bottom then left-to-right of a BVH, rather than the BVH itself. The method requires building a fixed-order traversal route through the scene. While leading to a stackless stream-like traversal, the method generally does not account for the ray direction and can have pathological cases where the mechanism traverses for a long time [95]. The method by Carr [53] introduced building a BVH for geometry images, using the Link-Map method, which enabled GPU ray tracing of rasterized geometry.

3.1.5 Sparse Matrix Tree

The sparse Matrix Tree method presented by Andrysco [12] is a novel way to efficiently store trees without memory pointers. The method can be thought of as placing a regular grid over the space. However, as the data is stored sparsely, cells with no data do not require any space in the tree. In addition, index-based referencing of nodes and node children is used in the method, allowing for stackless traversal. The research by Andrysco [12] was published after the work undertaken and published in this thesis.

3.2 Volume Visualization

Volume visualization plays an important role in modern science, medicine and industry. From the exploration of gaseous phenomena [340] and the analysis of industrial CTs [116], to virtual endoscopy of the sinus [168]. Visualization of volumetric medical data has seen much research over many years. The two prominent approaches typically employed are direct volume rendering [306, 90, 240, 38] and isosurface visualization.

3.2.1 Isosurface Visualization

Isosurface rendering is a simple and effective approach for the visualization the different intensities present in the data. Many methods for isosurface visualization have been presented ranging from direct volume rendering using isosurface transfer-functions [240], rasterization of isosurfaces extracted to polygonal data [197], or direct ray tracing [312, 165, 161].

Implicit kd-trees were presented by Wald [312] for single and multiple isosurface ray tracing on CPUs. The *implicitness* of Wald's method is that the kd-tree is actually built over the entire space and all possible values. As such the tree is an implicit acceleration structure for each isosurface, by storing the minimum and maximum value present within a node and its sub-branch. During traversal each node is tested against the desired value to see whether a ray-isosurface intersection will be possible within the node's region. In addition, implicit kd-trees were shown to be useful in the work by Grob [107] for Maximum Intensity Projection rendering. Wald further explored the use of an implicit acceleration structure with implicit BVH for tetrahedral grids [311].

Recently Knoll [162] introduced a peak-finding algorithm for Direct Volume Rendering. This method is useful when very narrow peaks in the transfer function are present, such as when a single isovalue is highlighted for isosurfacing. The approach is important in the sense that it is now possible to accurately render isosurfacee, in combination with DVR, in a robust and comprehensive manner.

An interesting method for isosurface rendering is the representation of volumes as Bezier Tetrahedra, as reported by Kloetzli [157]. The method involves dividing a cuboid volume into a tetrahedral grid and using BT to define density within each grid point. The BT volume allows any isosurface to be rendered quickly and at higher-quality (especially when super-sampling) than standard representation. The work was also implemented on a GPU for interactive framerates.

Object-order methods for isosurface visualisation have also been explored recently for GPU-based solutions. For example, Liu [193] presented a cell-projection method on modern GPUs, while in further work Liu [194] presented a multi-layer depth peeling approach for isosurface raytracing using single-pass hardware rasterization. Marroqium [209] also accelerated cell-projection using GPU hardware for DVR and isosurface rendering.

3.2.2 Illustrative and Context-Preserving Volume Rendering

Illustrative volume rendering does not attempt to produce realistic renderings of volumes. Rather, the main goal of illustrative and non-photo realistic volume rendering is the transfer of additional visual information to the observer. Basic approaches modify the local shading model; for example the Phong-Blinn [28] lighting model. The simplest and most known method is tone shading, an example being the work by Gooch [103]. Cartoon shading [179] attempts to mimic the artistic style of cartoonists. More complex modifications map information, such as the dot-product, to programmable look-up tables, such as the work by Bruckner [39].

Context-preservation typically modulates the opacity and colour of volume samples during the integration of rays by the gradient, the distance from the screen, the shade due to light and the accumulated opacity of the last sample along the ray. The approach they described was implemented on graphics hardware. Typically graphics hardware has limited memory to store information. As such their proposed method by fell short of being able to use the desirable curvature information of the volume samples and instead utilised what was already available to them on graphics hardware. Namely these were gradient and shading intensity. Using these two quantities they simulated the curvature.

Zhou [339] introduced a focus point and radius to DVR that allows a user to specify a region of interest. The rendering then enhances the focus region by modulating the opacity of each sample during rendering. Kruger [169] extended the work by introducing curvature into the focus region. By choosing the greatest opacity, between distance modulation or curvature modulation, they were able to have a focus region whereby the user would be able to 'see' into the volume but also get context cues because of the curvature. However, this method was localized and outside the region everything had high opacity.

Other methods for providing illustrative rendering include the work by Rautek [254] who used fuzzy logic and interaction-dependent rules to allow for more expressive illustrative rendering in real-time, while Rezk [258] provided opacity peeling to view through surfaces at the anatomy beneath.

3.2.2.1 Demand driven Raytracing

Demand driven [255, 245] ray tracing divides the complex tasks involved in raytracing to separate processes. This takes into account that parallel computing devices operate fully when performing the same task. The cost of such systems is additional workload to organize rays and the work queue prior to parallel computation.

The work by Foley [94] utilized multiple GPU shaders, where each shader was tasked with a separate stage of the ray-tracing pipeline; down-traversal, returns, shading, etc. This approach, i.e. rather than having a single shader for the entire pipeline, appears to be due to the limitations of GPUs at the time. Nonetheless it details a system whereby kernels are utilized in a demand driven manner, when needed.

3.3 Volume Segmentation and Classification

Volumetric data is usually comprised of intensity values. Segmentation and Classification are used to give meaning to data-values, or local features such that they can either be rendered differently or extracted from the data. Typically, classification is the real-time mapping of data-values, or derived attributes, to alternative colour and opacity contributions during visualization. Segmentation on the other hand is the separate task of defining a complex shape using a tool, such that it can be stored, extracted or used later to enhance the visualization. In effect, segmentation and classification are two sides of the same coin.

3.3.1 Segmentation

There are many methods that can be used for segmentation, as detailed in the previous chapter. However, this thesis focuses on using machine learning methods, specifically support vector machines, and GPU-based acceleration of segmentations.

3.3.1.1 Machine Learning Approaches

Machine learning methods are based on learning a set problem and then trying to predict whether new information matches the learned pattern. Both Neural Networks and Support Vector Machines are machine learning methods. An example of a Neural Network for segmentation was the work by Tzeng [301]. The work utilized both Neural Networks and Support Vector Machines (SVM) to deliver a method to classify higher-dimensional data. Their work incorporated a GUI, based on earlier work [300], where the user could paint training data onto the volume. This paint information was then converted into the higher-dimensional input vectors needed to train the learning system. Once trained, it was used to predict the classes of the remaining volume data. The SVM used in Tzeng's [301] work was implemented on CPU and as such suffered from slow classification of the volume. In addition, the SVM was trained after all input was gathered. This required that if input was changed the SVM had to be completely retrained with all inputs. Another method, presented by Song [287], utilized a probabilistic neural network for the partial segmentation of Brain MRI's. The approach also included a soft-labeling method to allow supervised learning using user input.

Further uses of SVM for segmentation and pattern recognition have been reported in the survey papers of Bryn [46] and Hai [119]. Of interest is the work by Lempitsky [188], who utilized a random forests technique for SVMs to allow for the real-time automatic segmentation and quantification of anatomical features; specifically accurate delineations of the myocardium present in real-time 3D echocardiography. Other uses of random forest SVM were reported by Statnikov [289]. Finally, Zawadzki [335] applied a support vector machine algorithm for the segmentation and visualization of retinal structures in volumetric optical coherence tomography data sets.

3.3.1.2 GPU-based Segmentation

With the advent of GPU programming languages researchers have utilized the computational power of GPUs to accelerate segmentation methods, as reported by Hadwiger [115]. For example, Lefohn [187] presented an interactive deformation and visualization of level-set surfaces that was 10 to 15 times faster than a CPU-based alternative. In addition the work provided a detailed user study to show the ease at which users could segment anatomy and effectiveness compared to a ground-truth manual segmentation. In similar work, Cates [54] accelerated level-set based segmentation using a sparse level-set GPU solver. Their work also provided an interface to allow hand-contouring to direct the segmentation. With the use of an ATI Radeon 9700 Pro, results showed a 10 fold increase in performance when compared to a CPU-based solver.

An important milestone was the work by Sherbondy [278] who presented a segmentation algorithm based on seeded region growing, accelerated with the use of GPU shader languages. The results for completing a segmentation of a 128^3 volume was approximately 6 seconds. The work was carried out on an ATI Radeon 9800 Pro and is notable for having a centralized structure, whereby the segmentation and volume visualization where carried out directly on the GPU, without the need to transfer any data from the GPU to CPU. The approach also allowed the real-time visualisation of the progression of the segmentation as the regions grew.

In a recent paper Chen [59] presented a tool-kit for volume sculpting and segmentation accelerated by GPUs. The work is interesting in that the took-kit, using seeded region-growing and region-shrinking methods, allowed a user to interactively, by way of a paint tool, adapt a segmentation in a variety of ways. For instance, the example presented in the work was to allow a user to peel away portions of the skull by painting a region on the screen first. Then the user could explore within the head, through the hole on the skull, and perform further segmentations on the brain itself.

3.3.2 Classification and Transfer Function Design

Classification and Transfer function design enables users to interactively explore volume data by allowing customisation of the rendering process. For example, Salma [268] presented a high-level user-interface for transfer function design, which allowed experts to design transfer functions for difficult tasks, but allow non-experts to easily adjust. This method was shown to be particularly useful in exploring anatomical features without the complexity normally associated with such a task. Other methods include the work by Zhou [339] who applied distance-based enhancement to volume rendering to allows the user to specify a region of interest and highlight that region during the visualisation.

In other work, Caban [47] utilized texture-based transfer functions, whereby a user specifies several textures with unique features, such as a portion of the brain. The input images are then analyzed and a transfer function is automatically designed to visualize similar features within the volume. This approach was shown to be able to classify regions of interest without prior knowledge of the specific of each volume.

Scale-space was utilized by Lum [201] for a novel classification system and user interface. The method transformed the volume in 4D scale-space. The work utilized filter banks and a novel interface to allow the user to easily control the classification process during rendering.

Coerrea [70] introduced size-based transfer functions. The work involved creating a scale-space of the original volume and tracking extrema. Relative sizes of features in the original volume could then be computed. Subsequently, transfer functions based on this information could automatically visualize the anatomical features present in the original volume, based on their size.

A method to automatically setup multi-dimensional transfer functions was presented by Roettger [260]. This was achieved by adding spatial information into a histogram and then classifying the histogram to form a transfer function with unique colours assigned to each class. Further, the work allowed user interaction with the ability to select classes in the histogram and customizing their properties. In addition, the work compared to other methods such as segmented volume rendering and showed in its ability to automatically highlight tumours with the proposed spatial transfer functions.

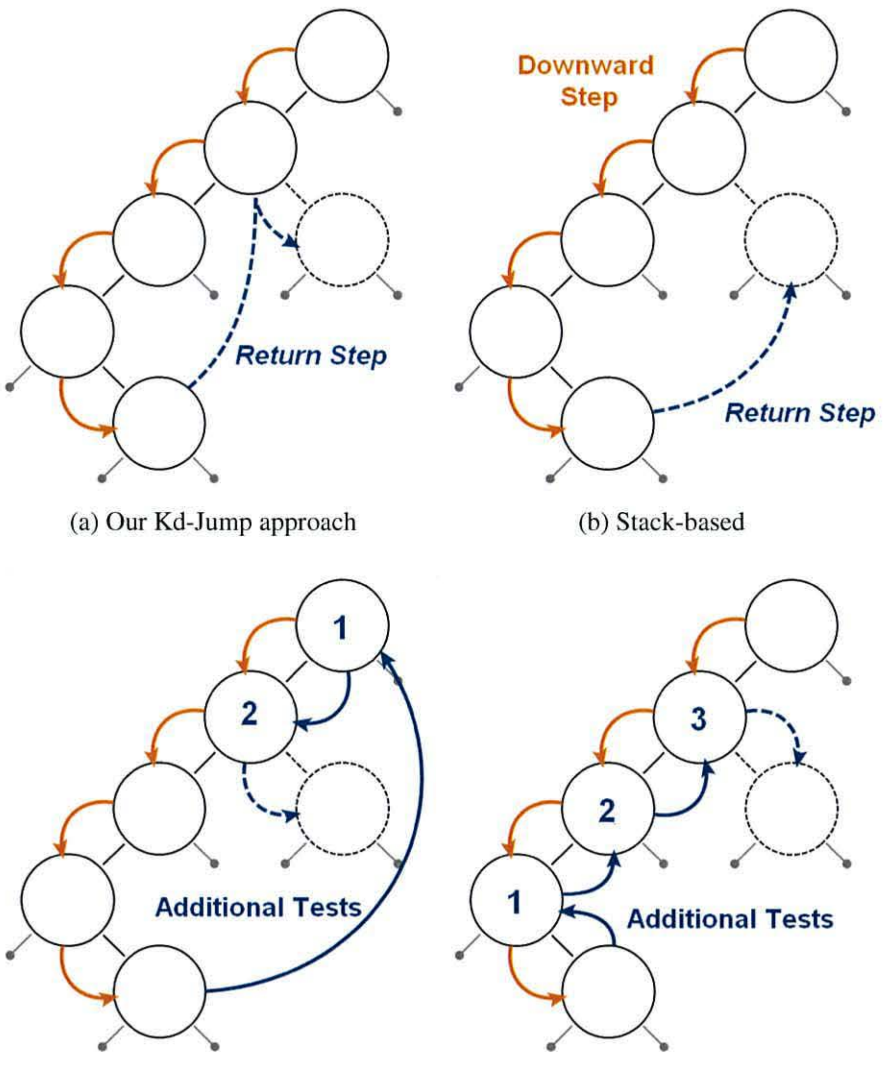
A number of recent advancements in the segmentation rendering have been made. Rendering of either the segmentation volume alone, or in combination with the intensity volume allows for improved visualization of specific anatomy. Hadwiger [114] renders segmentation data on consumer hardware and applies filtering of object boundaries. This work enabled high-quality, pre-integrated classification of segmentation objects without the need for the original intensity volume. Weber [318] renders segmented data using specialized transfer functions, where segmentations are defined by a contour-tree. The contour-tree contained detailed information on the topology of the volume.

A shape-based approach in extracting and rendering thin structures, such as lines and sheets, from three-dimensional volumetric data was presented by Huang [136]. Lee [186] developed a precise 3D image processing method to discriminate clearly the edges of segmentations by employing entropy maximization. The work by Kadosh [144] dealt with the reconstruction of segmented data, by applying a tricubic filter on distance fields. Finally, Hadwiger [118] rendered high-quality implicit surfaces on regular grids, for example, distance fields or medical CT scans, in real-time and could be applicable for segmented data.

Chapter 4

Kd-Jump

In this chapter a stackless traversal approach is presented. This approach is referred to as kd-jump, and is designed to traverse an implicit kd-tree to achieve an immediate return, much like a stack, without additional redundant node testing; as illustrated in Fig. 4.1. Also implemented is an enhanced form of the implicit kd-tree by Wald *et al* [312] where child nodes are tested prior to traversing into them. This has the potential to remove two iterations if a child is invalid; a down-traverse step and a return step. Finally, a hybrid kd-jump algorithm is presented, which utilises a volume-stepper for leaf testing and a run-time depth threshold to define where kd-tree traversal stops and volume-stepping occurs. By using both methods the benefits of empty-space removal and fast texture-caching can be attained.



(a) Our Kd-Jump approach (b) Stack-based

(c) Kd-restart (d) Kd-backtrack

Figure 4.1: Schematic illustration of additional nodes tested to achieve a correct return among different approaches. Kd-Jump does not need to re-test previously visited nodes.

4.1 Background

This work employs and builds upon the implicit kd-tree by Wald, *et al* [312]. This section details how an implicit kd-tree is formed and traversed.

4.1.1 Building Implicit Kd-Tree

A kd-tree is a binary tree where, starting with a root, each node is divided into two children by an axis-aligned splitting-plane. When a node cannot be split, it represents one voxel and is referred to as a leaf. The split axis is typically chosen based upon which dimension is currently largest for the node, as illustrated by Fig. 4.2.

Implicit kd-trees are required to be *balanced-trees*, such that all leaves of the tree are on the same depth. To achieve this balance, the voxel dimensions must be a power-of-two, although each dimension need not be identical. Implicit kd-trees define *actual*-dimensions and *virtual*-dimensions; where the actual-dimensions are for voxels that actually exist while the virtual-dimensions are used purely to ensure that a balanced kd-tree is built, as illustrated by Fig. 4.2. Even though the kd-tree is built upon a larger virtual-volume, the non-existent nodes and voxels are never visited; nor are they stored.

There are two stages to building the implicit kd-tree, both of which are iterative processes. The first stage involves determining the number of levels for the tree, computing the level information from top-to-bottom and allocating the node memory per level. The second stage involves calculating the node information from bottom-to-top. The initial building process is required to be run on the CPU in order to allocate GPU memory, while the more labour-intensive job of computing the node information can be performed in parallel on the GPU itself.

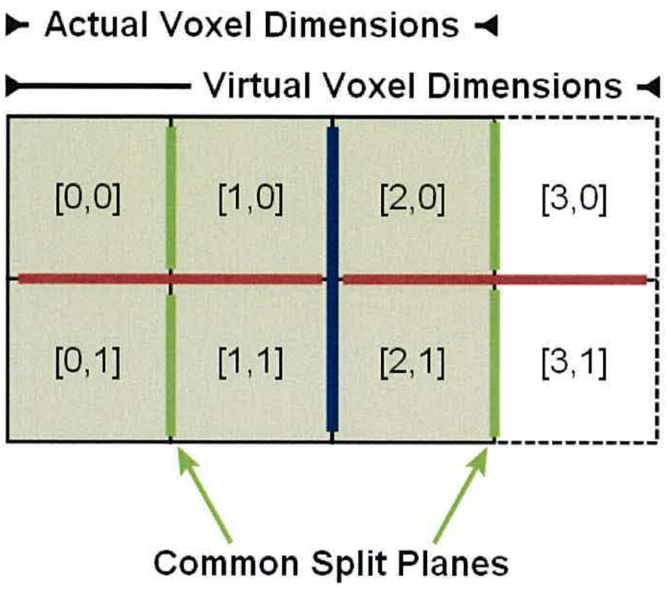
4.1.1.1 Initial Building

Given a volume and its actual voxel dimensions $R = [R_x, R_y, R_z]$, we first compute the virtual dimensions $V = [2^m, 2^n, 2^p]$ where $2^{m-1} < R_x \leq 2^m, etc.$ The number of levels in the kd-tree is then defined as $k = m + n + p$.

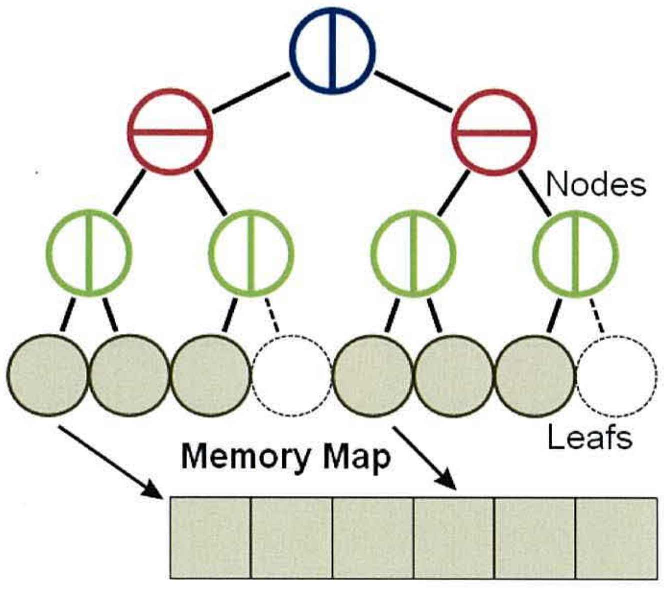
Each level of the kd-tree has real dimensions R^l and virtual dimensions V^l. During the process of tree building, the algorithm also needs to maintain the current range of nodes as $\widehat{V}^l$. We use $\widehat{V}^l$ to determine the largest dimension and the split takes place along the axis $a^l \in \{x, y, z\}$ of the largest dimension on level l. Thus, starting with $\widehat{V}^0 = V$ and $V^0 = [1, 1, 1]$, the virtual-dimensions for each level are defined by $V_{a^l}^{l+1} = 2V_{a^l}^l$, while $\widehat{V}_{a^l}^{l+1} = \widehat{V}_{a^l}^l/2$. The actual-dimensions of each level can then be found by $R^l = ceil\left(V^l\left(R/V\right)\right)$. Finally, the number of nodes which are required per level is $M^l = R_x^l \times R_y^l \times R_z^l$.

Unlike general kd-trees using memory pointers, nodes are addressed using indices and a map is used to convert the indices to a location in memory. For three-dimensional data, a node has three indices $U = [x, y, z]$, which are non-negative integers. Converting these indices, for a node on level l, to a memory location is achieved with $offset + (U_x + (U_y \times R_x^l) + (U_z \times R_x^l \times R_y^l)) \times sizeof(node)$, where the $offset$ is the start of data for the level.

Implicit kd-trees do not store a split-plane within each node. For each level of the kd-tree, the number of shared split-locations is $R_{a^l}^l$ rather than M^l; see Fig. 4.2. In fact, the split plane for a node can be computed, for any node, during traversal so as to avoid using global memory.



(a)



(b)

Figure 4.2: Wald's, *et al* [312] implicit kd-tree. A balanced kd-tree is formed by building the tree upon the virtual-voxel dimensions, while only actual nodes and leaf data are stored. Also, note that split planes can be shared.

4.1.1.2 Computing Node Data

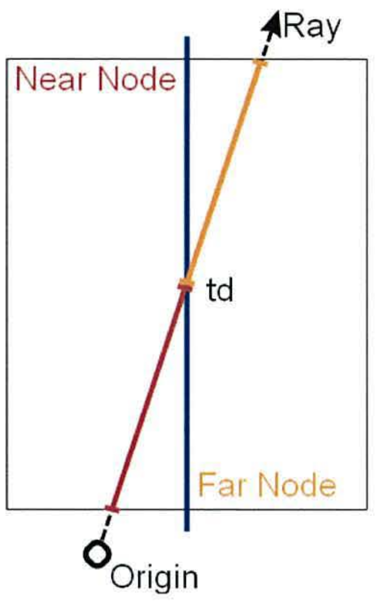
Once the memory for the kd-tree is allocated, the node data and dimension splits for each level can be computed. Wald, *et al* [312] defined that each node contained the minimum and maximum (min/max) value for all data held within the node sub-tree. This min/max value is used to determine whether any child contains the current iso-value and whether traversal should continue.

This thesis defines that each node contains the two sets of min/max values; one for each child; rather than just one min/max value encompassing both. By storing the min/max of both children, faster traversal can be achieved. Specifically, the method can store both sets so that referencing of the data can be achieved by mapping the indices to memory once and loaded in one transaction. In addition, by checking whether both children are valid (contains the isosurface) before traversing into them, we can potentially eliminate two redundant iterations (down-traversal and return) if a child is invalid.

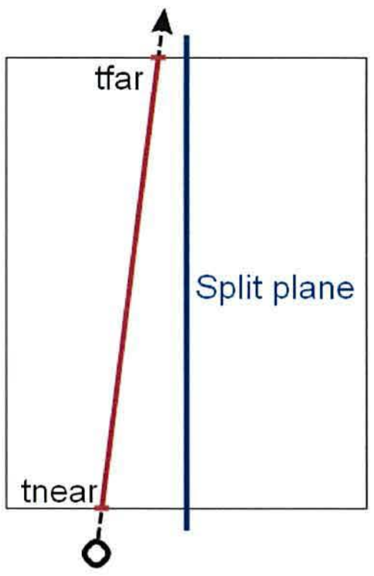
Starting from the last node level, the min/max values are computed by evaluating the children. In the case of the last level, this requires checking the eight corner values of a voxel. For the remaining node levels, the min/max sets are computed from the min/max sets held by the child nodes. As nodes are only dependent on their own children, all nodes on a tree-level can be computed in parallel. Finally, if memory overhead is a concern, the final level of nodes can be omitted and computed on the fly [312]. Special care must be taken if the volume is accessed via CUDA textures, as data is typically aligned to texels, rather than voxels; i.e., the data value is at the centre of a volume cell, rather than at the corners.

4.1.2 Traversing Implicit Kd-Trees

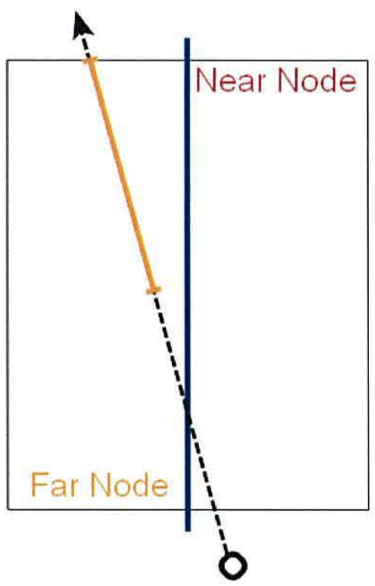
Determining whether a ray intersects the isosurface is achieved by traversing the nodes which both intersect the ray and contain the isosurface. Starting from an origin, a ray is projected along a direction and a ray segment, defined by t_{near} and t_{far}, is used to mark the valid portions along the ray where ray tracing can occur. Each node has two children, denoted *first-child* and *second-child*. During traversal, the children are also tagged as *near-node* and *far-node*, although the conventions are not synonymous. Like Wald, *et al* [312], a boolean $NearFirst \equiv r_{a^l} > 0$ is defined, where r is the ray direction vector. Traversal from the parent into a child node is performed by updating the indices; we update $U_d = 2U_d + (1 - NearFirst)$ for the near-node and $U_d = 2U_d + (NearFirst)$ for the far-node. By traversing the near-node initially, we ensure that the first intersection along the ray is found, at which point traversal can terminate.



(a) $t_{near} < t_d < t_{far}$



(b) $t_{far} < t_d$



(c) $t_{near} > t_d$

Figure 4.3: The three cases of traversal. (a) the ray-plane intersection distance t_d is within the ray segment (t_{near}, t_{far}), (b) the ray segment lies completely on the near side of the split plane, (c) the ray segment lies completely on the far side of the split plane.

Testing a node initially requires computing the intersection distance t_d from the ray origin to the split plane. Determining which children to traverse into is subject to where t_d is in relation to t_{near} and t_{far}, as shown in Fig. 4.3. If $t_{near} > t_d$ then the near-node is traversed and the far-node is culled. If $t_{far} < t_d$ then the far-node is traversed. A common case is when $t_{near} < t_d < t_{far}$ and both child nodes are valid and (potentially) must be traversed. In this case, the near-node is initially traversed into and, if the ray does not find a valid intersection in that sub-tree, the algorithm returns to the far-node.

The typical solution for storing the *far-node* is to use a stack to record the indices and the t_d and t_{far} values. However, a stack is not ideal for use on a GPU and this thesis explores a stackless approach.

4.2 Stackless Traversal with Kd-Jump

The basic traversal method of the traditional kd-tree is to store a return point when both children of a node are valid. Utilising a stack is a simple method to store the return information. However, a stack approach requires that the (currently) slowest part of the GPU pipeline is utilized; i.e., the global-memory. To avoid using the global-memory, one must remove the stack. There are two main stackless methods currently available (without requiring additional node memory); kd-restart and kd-backtrack. Both are trivial to implement for implicit kd-trees, but lead to additional workload compared to a stack-based approach. The extra work comes in the form of redundant node testing to find a continuation point.

With kd-backtrack, the return mechanism is replaced by traversing back up the tree node-by-node until a valid node is found, at which point downward traversal continues. Once a valid parent node is found, the *far*-child is traversed. The approach was originally envisaged for use with arbitrary kd-trees and therefore required additional parent-pointers to work.

Traversal of implicit kd-trees does not involve memory pointers and all nodes for all levels are entirely referenced by indices. As a result, it is possible to forgo the need to backtrack one node at a time and simply jump immediately to the next valid node. A novel approach for this is now explained and referred to as Kd-Jump.

4.2.1 Traversing to Child

Traversal of the kd-tree involves tracking and updating three indices, so as to allow addressing of nodes. The indices of a node, at level l, are defined as $U^l = [x^l, y^l, z^l]$ and the indices of the next level are U^{l+1}. The algorithm first initializes the child indices with those of the parent; $U^{l+1} = U^l$. Then traversal, from parent to child, is achieved by altering the index-component corresponding to the axis a^l that splits the l'th level

$$U_{a^l}^{l+1} = 2U_{a^l}^{l} + c, \quad c = \begin{cases} 0 & \text{first child} \\ 1 & \text{second child} \end{cases} \tag{4.1}$$

4.2.2 Returning to Immediate Parent

Like a stack-based approach, the best scenario is to return to the next immediate node to test. The current node and the node to return to will always share a common parent. As such, the first step is to arrive at that parent (see Fig. 4.1(a) or Fig. 4.4). The trivial case, given U^{l+1}, is to return to the immediate parent U^l. Again, for this simple case, the algorithm initializes the indices with $U^l = U^{l+1}$ and then apply an operation equivalent to the inverse of Eq.(4.1).

$$U^{l}_{a^l} = floor\left(\frac{U^{l+1}_{a^l}}{2}\right) \tag{4.2}$$

4.2.3 Returning to Arbitrary Parent

Returning from U^l to U^j $(j < l)$ potentially requires different divisions of each element of the index set. To achieve an immediate jump, we must deduce the number of iterations of Eq.(4.1) that have been performed to each element of the index set.

We define a k-by-3 matrix $\mathbf{S}$, which stores the accumulation of the number of axis-splits, for each level. The matrix is formed in a recursive manner. Each row is initialized with the previous row values; $\mathbf{S}_{l+1} = \mathbf{S}_l$. This is then followed by altering the vector-component corresponding to the axis a^l that splits the l'th level

$$\mathbf{S}_{l+1,a^l} = \mathbf{S}_{l,a^l} + 1, \tag{4.3}$$

where $\mathbf{S}_{m,n}$ represents the matrix element at the m'th row and n'th column, and

$$\mathbf{S}_0 = [0,0,0] \tag{4.4}$$

is the initialization vector for the root level. Note that $\mathbf{S}$ is formed only once during kd-tree construction. Thus, storing and accessing this matrix on GPUs can be made quite fast by using cached constant-memory.

Given the accumulation matrix $\mathbf{S}$, the current depth l and the depth j of the common-parent node, we can find the numbers of iterations, denoted N, of Eq.(4.1) applied to each index element between levels j and l as

$$N = \mathbf{S}_l - \mathbf{S}_j \tag{4.5}$$

where $N = [N_x, N_y, N_z]$.

Finally, the algorithm is able to restore the index-set for the parent node being returned to by altering Eq.(4.2) to acknowledge the number of power-of-2 multiplications that have been applied (Eq.(4.5)). Thus, returning to the index-set U^j, given U^l, is achieved using

$$U^j = floor\left(\frac{U^l}{2^N}\right), \tag{4.6}$$

where $2^N \equiv [2^{N_x}, 2^{N_y}, 2^{N_z}]$. So long as $c \leq 1$, Eq.(4.6) will correctly find the integer indices without having to re-determine c for each level. Also note, all divisions in Eq.(4.6) are a power-of-two and, therefore, they can be implemented using rightward bit-shifting.

4.2.4 Completing Jump

Once we have returned to the parent node, we can simply reapply Eq.(4.1) in order to traverse into the next child. However, the unknown element is the offset c, which we need to apply in order to arrive at the *far-child*. Assuming a *nearest-first* traversal ordering, this can be deduced by redetermining whether the *first-node* is the *near-facing node*, which is quickly performed by examining the ray direction; such that if $r_{a^l} \geq 0$ then $c = 1$ else $c = 0$. The complete Kd-Jump method is illustrated by Fig. 4.4 for a simple two-dimensional case.

After returning into the next node, the final step is to re-clip the ray to the bounds of the node and recompute the *t-near* and *t-far* intervals. The bounds can be computed on-the-fly, which avoids global-memory usage as well. See the work by Williams [327] for efficient Ray-Bound intersection methods.

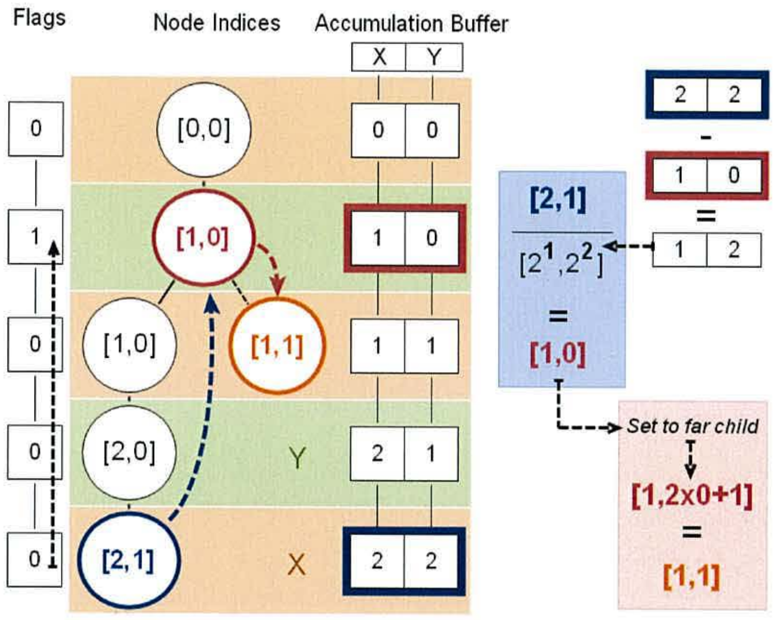
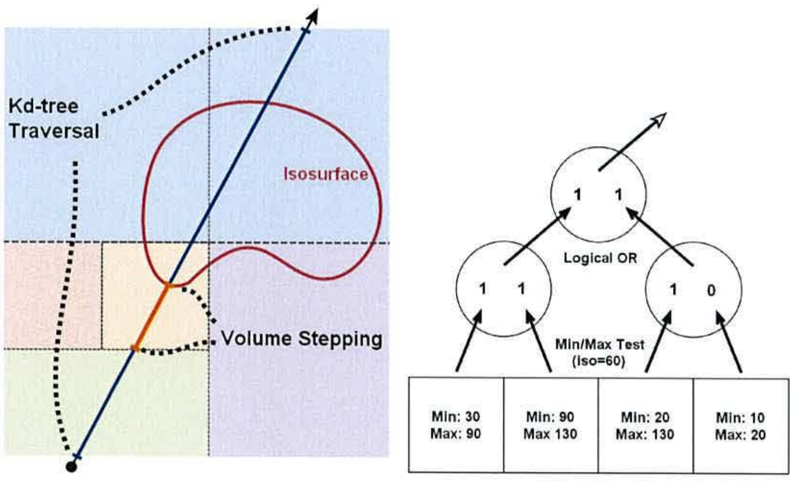


Figure 4.4: A two-dimensional example illustrating the two-stage process of finding the indices of the next node to test.



(a) Hybrid Traversal (b) Dynamic Update

Figure 4.5: With hybrid traversal, the kd-tree is traversed until a variable threshold is met after which volume stepping occurs. With Dynamic update, nodes are updated with binary conditions as to whether children contain any part of the target isosurface.

4.2.5 Making Return Flags

Our detailed Kd-Jump mechanism facilitates a return, but does not yet include how much to jump by. To return immediately during the course of ray traversal, we require a *flag* to specify whether a level along the traversal path has a node that requires testing. One should note that, for the current traversal path, only one possible node will be required to be returned to for each level. As such, we only require a single memory-bit per level to store a possible return. We define a 32-bit integer-register $DepthFlags$ to store these flags. As the typical size of volumes used on GPUs today (without out-of-core methods) is less than 1024^3, a 32-bit integer can hold the depth-flags. However, 64-bit integers can be utilized to facilitate kd-trees of up-to 64 levels in the future; indeed CUDA devices already provide 64-bit hardware functionality.

Given the $DepthFlags$ register, we set whether a level should be returned to using *bitwise operators*; $DepthFlags \mid= 1 \ll (31 - l)$. Note that we store the bits in *most-significant* ordering, such that the bit-index of the l'th level is $31 - l$. We can determine whether there are return positions by checking if $DepthFlags > 0$. Assuming that $DepthFlags$ is non-zero, finding the first-set depth flag is akin to counting the consecutive number of zero-bits, starting from the *least-significant* bit; we denote this operation $CountBits$. Hence, the actual depth j to return to is $31 - CountBits(DepthFlags)$. Upon a successful return to a level, it is important to clear the j'th level flag bit to zero; again using bit manipulation. In CUDA $CountBits$ can be accomplished using the built-in function ffs (however it is offset by plus 1). For an alternative to CUDA's ffs, see Andersons's 'Bit Twiddling Hacks' [11].

4.3 Faster Traversal with Hybrid Kd-Jump

An acceleration method for ray-tracing serves one primary purpose of removing the extraneous memory access affiliated with empty or invalid space. However, it is entirely possible that an acceleration method might under-perform or even perform worse than a brute-force ray tracer, for example, when the acceleration method is complex or is utilized for too long. Thus, it is very important to be able to determine when and where an acceleration method is useful. For this purpose, we present hybrid traversal and dynamic update.

4.3.1 Hybrid Traversal

Each node in a kd-tree represents a sub-region of the complete volume. With each level of the kd-tree, this region is made ever smaller until a node represents a single voxel on the final level. This thesis employs a simple method, whereby we introduce a real-time depth threshold parameter to the traversal kernel. Once rays traverse past this threshold, we switch to the volume stepper and iteratively step along the ray from t_{near} to t_{far}. The volume stepping is performed until the isosurface is crossed, or until t_{far} is reached, after which a return is issued. Fig. 4.5(a) depicts where volume stepping within a volume region is used after traversal of the implicit kd-tree.

The purpose of this hybrid system is two-fold; firstly, to gain the benefit of the fast texture-cache and, secondly, to allow the adjustment of the threshold in order to maximize the usefulness of the kd-tree. Although combining an acceleration structure with volume stepping methods is not entirely new [241], we present it here in order to show that a kd-tree can perform well and can be adjusted easily for dynamic situations. In addition, with a variable threshold, the point when the acceleration structure is useful and when it is not (slower) can be analyzed.

By building a complete kd-tree and then introducing a run-time depth threshold a user can alter the threshold, during ray tracing, in order to find the optimism setting. For instance, the optimal threshold is subject to several factors, primarily the volume size, the complexity of the data itself, the isosurface location and the view direction. Further, if the volume stepping distance is reduced, say to acquire better intersection results while zoomed in, then a larger threshold (traversing further down the kd-tree) would be more efficient.

Also, the threshold depends on the complexity of the data-interpolation being performed. If *tri-linear* interpolation is used, it would be more beneficial to switch to the volume stepper sooner. However, if *tri-cubic* interpolation is used, as is the case for discrete binary-volume rendering [144], then there is far more incentive for the kd-tree to traverse for as long as possible, because of the lack of hardware acceleration. The same argument applies for complex intersection methods such as the correct root finding method [208].

4.3.2 Dynamic Update

With the original implicit kd-tree work by Wald *et al*, each node contained a min/max pair; the minimum and maximum values within the region represented by the node. As described in the previous section, both child min/max pairs are loaded prior to traversal into children. Hence, during traversal, these two sets of values must be loaded from memory. For 8-bit data, this requires a 32-bit transaction while, for 16-bit data, the size of the node is 64-bits. The cost of loading this data, plus the cost of comparing the node value range with the target isosurface, may add additional complexity.

A better alternative is to move the node validity test out of the traversal stage and into the kd-tree update stage. Thus, instead of a node having two min/max pair's for each child, it simply has two boolean bits to specify if the children are currently valid for traversal. Upon a change in isovalue, this would require updating every node on every level starting from the original volume itself, as depicted in Fig. 4.5(b). This is already quite fast (less than 0.25 seconds) for 512^3 volumes, even with a naive implementation. Much of this efficiency can be attributed to CUDA's cached texture-access, which not only applies to accessing three-dimensional volume, but also accessing the node data.

With hybrid traversal, several deep-levels of the kd-tree may be avoided all together. It is shown, in the results, that the deeper levels are not particularly useful in our implementation. Therefore, the dynamic update can be made more efficient by only updating levels of the tree which may be traversed. This can be accomplished by introducing a separate sub-volume of min/max pairs. This sub-volume would represent the node information for a kd-tree level and would be considered the absolute cut-off depth. During traversal, the *cutoff* depth would have to override the depth *threshold* if the later is greater.

Choosing whether to employ node-conditions or traversal-conditions depends on several factors. If memory size is an issue, or the isovalue is changed irregularly, then node-conditions would be more suited. In contrast, if the isovalue is altered every frame, then traversal-conditions would be better suited.

4.4 Results

This work performed several comparative tests and recorded the timing information for the kernels using CUDA's high-resolution timers. The results presented here where averaged over multiple passes. Table 4.1 gives the results for the average frames-per-second (FPS) spanning a wide range of the isosurfaces and multiple view directions for the test data. Fig. 4.8 and Fig. 4.9 show the rendered results.

Table 4.1: Average FPS across multiple views and multiple isovalues. Bonsai, Foot and Skull are of 256^3 in size while Aneurism is of 512^3.

	512^2			1024^2		
	stack	kd-restart	Kd-Jump	stack	kd-restart	Kd-Jump
Bonsai	58.3	34.2	**65.6**	17.3	10.0	**18.9**
Foot	43.1	25.7	**48.8**	12.3	7.2	**13.6**
Skull	52.9	32.1	**59.7**	15.5	7.1	**16.8**
Aneurism	42.9	25.5	**50.2**	11.7	6.5	**12.9**

Memory usage is an important factor for GPUs, due to limited resources. It is shown the typical memory usage in Table 4.2 for a 1024^2 screen, as would be the case with single ray-tracing kernel. The table shows a stack requires considerable amount of global memory to accommodate all rays, while Kd-Jump requires only a small matrix in fast constant-memory. Although kd-restart uses the least resources, the redundant node visitation severely reduces performance as shown in Table 4.1.

Table 4.2: Memory usage (per kernel) for traversal schemes with 1024^2 screen resolution and maximum depth of 27. It assumes that CUDA will allocate all device memory for all threads (rays). Kd-Jump needs only a small amount of constant-memory (cached) and no device memory, while a stack requires a considerable amount of device memory.

	stack	kd-restart	Kd-Jump
Device	405MB	0	0
Constant	0	0	0.0003MB

To further compare the performance of Kd-Jump, this work evaluates the theoretical performance, as shown in Fig. 4.6. In this evaluation, this work only tests the relevant code to store and retrieve a return position. A kernel with 1024^2 threads organized into 128-thread blocks, was set up, to achieve full occupancy. Both the stack and Kd-Jump kernels were tasked with storing and then retrieving n number of returns. The result clearly shows that Kd-Jump potentially has considerable speed gains. When cross-referenced with Table 4.1, however, it is evident the theorized gains of Kd-Jump over stack, in a complete ray-tracer, do not show as great of a speed gain. It is believed that it is due to the fact the memory accesses in the stack kernel are hidden better by the other computation; i.e. the general traversal loop, ray splitting computations and leaf testing.

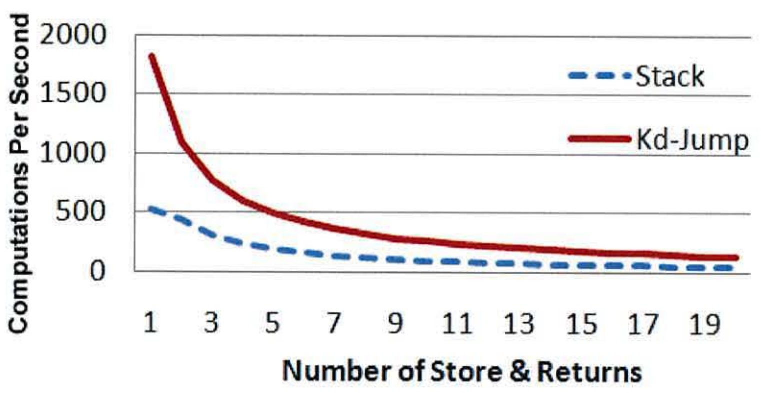


Figure 4.6: Theoretical performance (number of computations possible per second) between stack and Kd-Jump for increasing number of returns. 1024^2 threads perform n storages and then n returns.

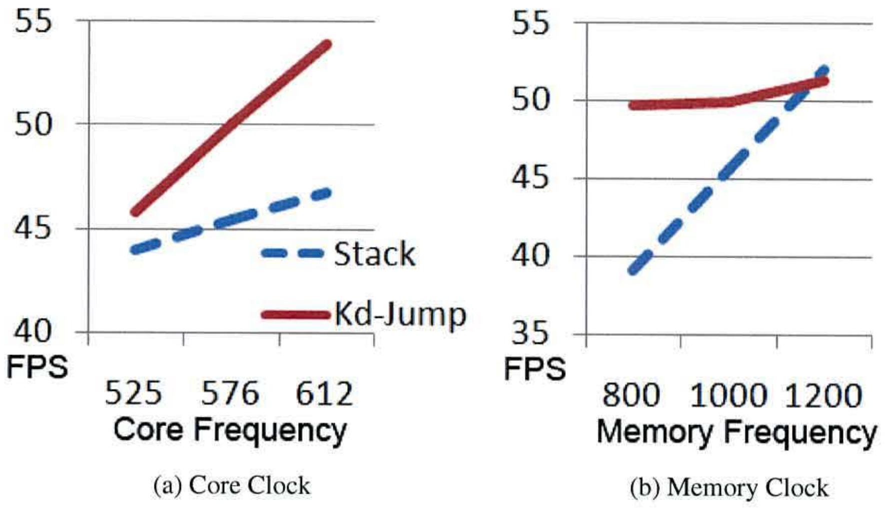
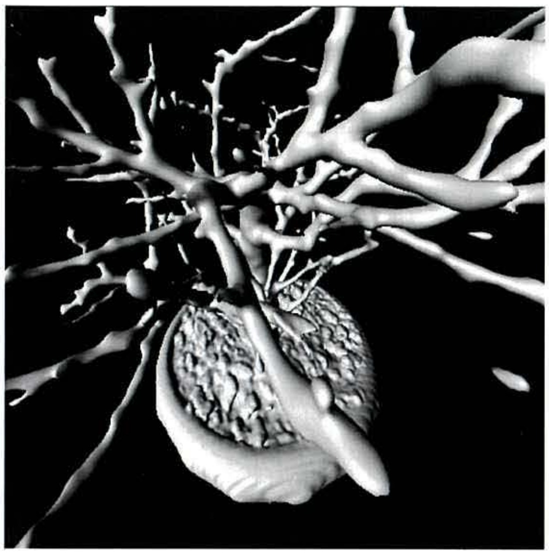
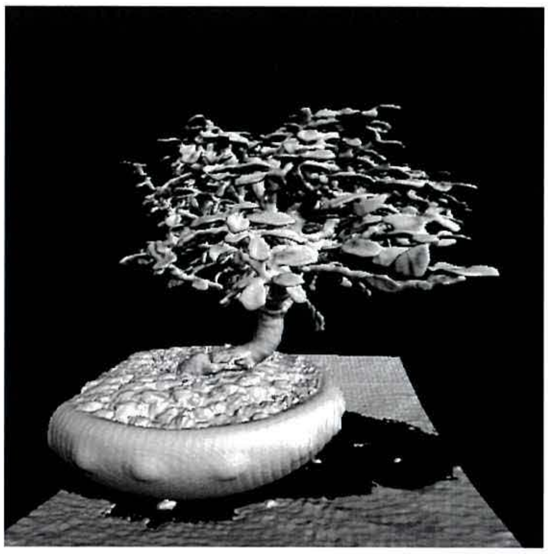


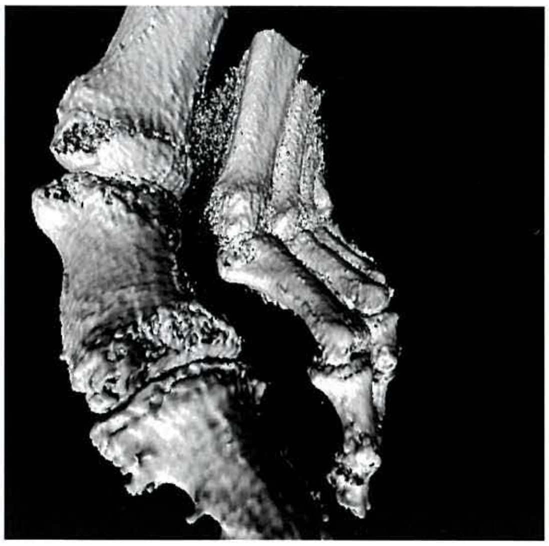
Figure 4.7: FPS for Kd-Jump and stack traversal with different core frequency and memory frequency settings, rendering Foot (256^3) with 512^2 resolution. It shows that Kd-Jump is computationally limited whereas stack is memory-latency limited.



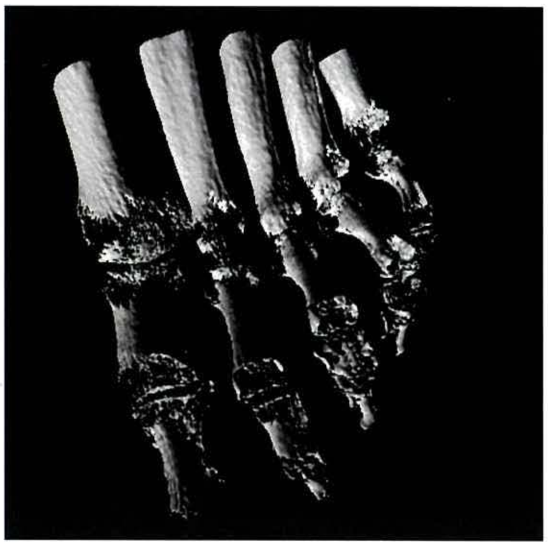
(a)



(b)



(c)



(d)

Figure 4.8: Isosurface rendering results of the (a,b) Bonsai (256^3) and (c,d) Foot (256^3) Datasets

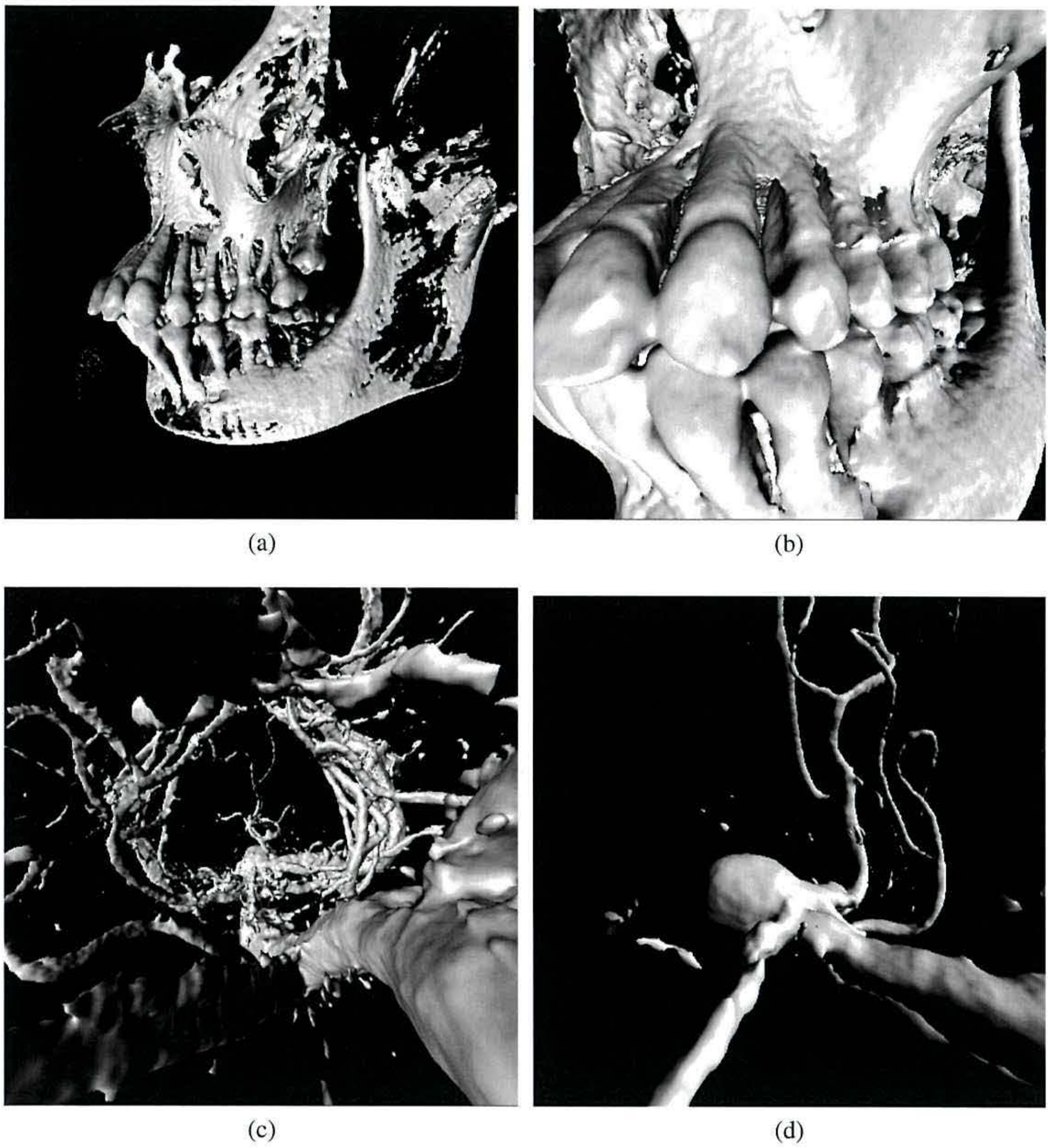


Figure 4.9: Isosurface rendering results of the (a,b) Skull (256^3) and (c,d) Aneurysm (512^3) Data-sets

4.4.1 Limiting Factors

We can test both the Kd-Jump and stack-based kernels with different settings for the core and memory frequencies, as shown in Fig. 4.7. This allows us to examine which factor (computation or memory access) is limiting performance for each kernel. The results clearly show that Kd-Jump is computationally limited while the stack-based approach is memory limited. This quick test can also be quite useful during development and implementation of ray tracing kernels, as it can indicate which factors should be optimized.

4.4.2 Hybrid Kd-Jump

In order to gain in performance as much as possible and thus give merit for using a kd-tree in the first place, a comparison of a hybrid kd-tree kernel (using the presented Kd-Jump method) versus a pure brute-force ray tracing kernel was preformed. While these kernels are not optimized particulary well, both share the same code for stepping through the volume and detecting an isosurface crossing.

For the Hybrid Kd-Jump kernel, this work incorporated a number of optimizations, specifically the hybrid traversal and dynamic update described in section 4.3.1 and 4.3.2. In addition, the Hybrid Kd-Jump kernel accesses node information from the texture cache rather than directly from global memory, which results in faster access.

Table 4.3 shows the results for Hybrid Kd-Jump versus a brute-force volume-stepper; showing multiple isosurfaces, data-sizes and screen sizes. For the sake of examining the effect of data-size on the performance, up-sampled (linear interpolation) versions of the bonsai, skull and foot data-sets were created, as well as a down-sampled (averaged) version of the Aneurism data-set; On average the rendering of 256^3 sized

Table 4.3: FPS for Hybrid Kd-Jump versus brute-force averaged across multiple views per isovalue. Hybrid threshold is chosen to maximize performance in each case. It clearly shows that Hybrid Kd-Jump outperforms a brute-force volume-stepper for most cases. The volume-stepping is faster only when there is very little empty space and the isosurface is found quite quickly (i.e., close to the bounds of the data).

			512^2		1024^2	
data	size	isovalue	brute	Hybrid	brute	Hybrid
Bonsai	256^3	40	**132.0**	123.0	**41.8**	37.3
		100	90.2	**130.3**	31.5	**38.0**
	512^3	40	36.3	**64.7**	16.9	**21.6**
		100	22.1	**81.6**	10.4	**26.3**
Foot	256^3	40	117.0	**126.6**	36.3	**39.0**
		100	78.5	**124.8**	27.0	**40.2**
	512^3	40	41.0	**71.1**	15.9	**25.3**
		100	21.1	**66.9**	8.8	**22.4**
Skull	256^3	40	**155.0**	100.3	**51.8**	30.0
		150	68.9	**261.2**	26.7	**75.4**
	512^3	40	41.0	**71.1**	15.9	**25.3**
		150	21.1	**66.9**	8.8	**22.4**
Aneurism	256^3	40	76.0	**126.1**	26.9	**34.5**
		100	67.6	**282.4**	24.1	**76.5**
	512^3	40	20.1	**53.2**	8.9	**16.1**
		100	16.0	**164.6**	7.1	**46.6**

volumes was twice as fast than the rendering of the 512^3 volumes. Also of interest, in Table 4.3, are the cases where brute-force outperforms Hybrid Kd-Jump. In these cases, two conditions are (always) present. Firstly, the isosurface covers much of the screen and secondly, the isosurface is close to the bounds of the data. Hence, a simple volume stepper only operates for a short period of time before detecting an isosurface. With more complex isosurfaces, longer distances from the bounds to the isosurface and larger screen resolutions, however, brute force is slower than Hybrid Kd-Jump. Fig. 4.11 shows the performance change for various threshold values and indicates the degree to which using a kd-tree is beneficial.

4.4.3 Multiple Rays Per Thread

A bottleneck affecting all methods can occur when only one thread of a warp is active, or only one warp of a block is so. If CUDA allocates a block worth of resources (shared memory, registers) and operates that block until completion, it is logical to assume that, if only one warp is actually active, then the three remaining inactive warps will actually limit computation throughput.

We can test this by altering the kernel to include an outer-loop, whereby new rays are loaded and initialized, once a warp has finished. It is doubtful that loading a new ray when a single-thread terminates will be effective. Indeed, during the initial development, loading a new ray when each thread terminated induced much slower performance. It is believed that this is due to the result of more code-branching during traversal, as well as the removal of the initial ray coherence. On the other hand, a warp terminates when all threads terminate. Thus loading a new batch of rays across the 32 threads of the warp will maintain the initial coherency of a group of rays while ensuring that as many warps are active throughout the lifetime of the kernel. This work implemented the multiray kernel as an extension of the Kd-Jump kernel. As seen in Fig. 4.12, we show positive benefits.

4.4.4 Separating Kernels

The basic approach to parallel ray tracing is to dedicate one thread per ray and to develop a single-kernel containing the entire rendering pipeline; node traversal, leaf testing and pixel shading. However, as a single-kernel, the pipeline will not fully exploit the GPU and may well indeed create performance bottlenecks. For instance, shading is a branchless process and hence should perform very well in parallel. In a single-kernel ray tracer, however, some threads may begin shading prior to others; this causes thread divergence and serialization events. Thus, separating out the shading portion (as well as other portions) of the pipeline and placing it in a different kernel should result in better performance, at least in theory [245, 255]. That being said, launching multiple kernels can carry an overhead. Fig. 4.13 shows this theory has at least some merit. However, the performance improvement is small and only attained if a lot of shading occurs to begin with; the rendering of lower isovalues occupy a large portion of the screen.

4.5 Discussion

CUDA devices contain multiple processing units. Each processing unit is capable of operating many threads in parallel, although only a small number (a warp) actually work at any given moment. Currently, CUDA devices operate warps of 32 threads in size. With branchless code, all threads in a warp operate the same instruction of code and fully utilize the SIMD (Single Instruction, Multiple Data) functionality. If conditional branching occurs, then the threads branching into the statement are evaluated first and any thread not following the branch is masked inactive and forced to wait. Once the branch is evaluated, a serialization occurs and threads in the warp are re-synchronized automatically. Apart from the fact that divergent branching and serialization incurs slowdown, the SIMD functionality might not be used to the fullest. Also, note that SIMD efficiency is dependent on limited code-branching and not necessarily ray-coherence.

Currently, a maximum number of 1024 threads can be active on each multi-processor. While only a single warp (group of 32 threads) ever works at any given moment for a multi-processor, CUDA is able to switch between warps waiting for instructions to complete and effectively ensure maximum throughput. For example, if one warp requests global memory, it will essentially have to wait for that request to complete and, during that time, other warps can operate. Thus, maximum occupancy ensures that costly instructions (such as memory access) are better hidden, by the GPu thread scheduler, and do not pose a bottleneck; this observation can be made by comparing Fig. 4.6 and Table 4.1 where additional computation better masks the memory latency. In Fig. 4.12, it is shown that further performance improvements can be gained with load-balancing (multiple rays per thread).

Concern for maximum occupancy should be the first priority for researchers. Device occupancy is determined by two factors, the number of registers and the amount of shared memory used by the kernel. Unfortunately, the limited amount of these factors makes full occupancy improbable to have the entire traversal pipeline as a single kernel (on current architecture). To achieve full occupancy, without extensive and time-consuming optimization, the traversal mechanism must be separated into multiple kernels.

4.5.1 Multiple Kernels versus Single Kernel

Separating kernels into multiple stages presents a new challenge; how should we organize the work for them? For example, let us assume that we have an Intersect-Kernel which detects ray-geometry intersections and a Shader-Kernel. Not all rays will have intersections and, therefore, they will not require shading. Cropping out the rays which do not require shading so that we can pass only valid rays to the Shader-Kernel requires an intermediate step to organize the memory.

A simple solution is to have the Intersect-Kernel to store a hit-flag specifying whether the ray has hit geometry. A separate kernel can then examine rays to find those with valid hits and create a work-list for the Shade-Kernel. This approach can also be performed in parallel (as a minimizing problem [106]), where each thread is responsible for checking the state of a set number of rays. Regardless of how it is implemented, however, the additional step to organize the work requires extra computation and memory access, and is therefore only useful if the benefit outweighs this cost.

Simply separating kernels *without* organization of the input workload, for the shading kernel (i.e., simply having 1 thread per pixel), is shown in Fig. 4.13. However, what remains to be seen is whether it is possible to reorganize workload between kernel calls, without it becoming a bottleneck in itself.

4.5.2 Alternative To Accumulation Matrix

The accumulation matrix approach, while simple, still requires constant-memory for storage. Different architectures may not provide fast caching features. An alternative to the accumulation matrix is to utilize more registers; one per-dimension. Each bit represents a level of the kd-tree. We store a TRUE, in the relevant register, to specify whether a dimension has been split on a particular level. The registers would be propagated with the correct split information during the kd-tree build stage, or actually during traversal.

Determining N of Eq.(4.5) using the accumulation registers involves counting the number of TRUE bits between the current depth and the return depth. Firstly, this requires masking the accumulation registers for only the levels in question and a bitcounter. In CUDA, this can be achieved using the *__popc* function (see Anderson [11] for an alternative using right-ward zero-bit counting).

4.5.3 Limitations and Scope of Kd-Jump

While Kd-Jump exploits the indexing method for implicit kd-trees, pointers are used for general kd-trees. As such, Kd-Jump in the current form cannot be readily used for a general kd-tree. In order to apply Kd-Jump to a general tree, one should be able to transpose a general kd-tree onto a 'virtual' balanced kd-tree and build a suitable memory map to reference node data. In practice, however, any additional computation for the map could lower the performance.

When Kd-Jump is employed for isosurface ray-tracing or direct volume rendering, the traversal-orders are pre-defined and are quickly recomputed upon return. For MIP (Maximum Intensity Projection) rendering, however, re-determining the traversal-order would require additional memory look-ups and testing, which could lower the performance.

The Kd-Jump approach would be applicable for use with other binary trees, if nodes can be referenced with indices and index-updates can also be invertible. In theory, this approach could be used with BVH [184] if indices are employed and a sufficient method for mapping the indices to memory (without excess) is available. Since a BVH is not a spatial-splitting structure, tree-balancing is applicable.

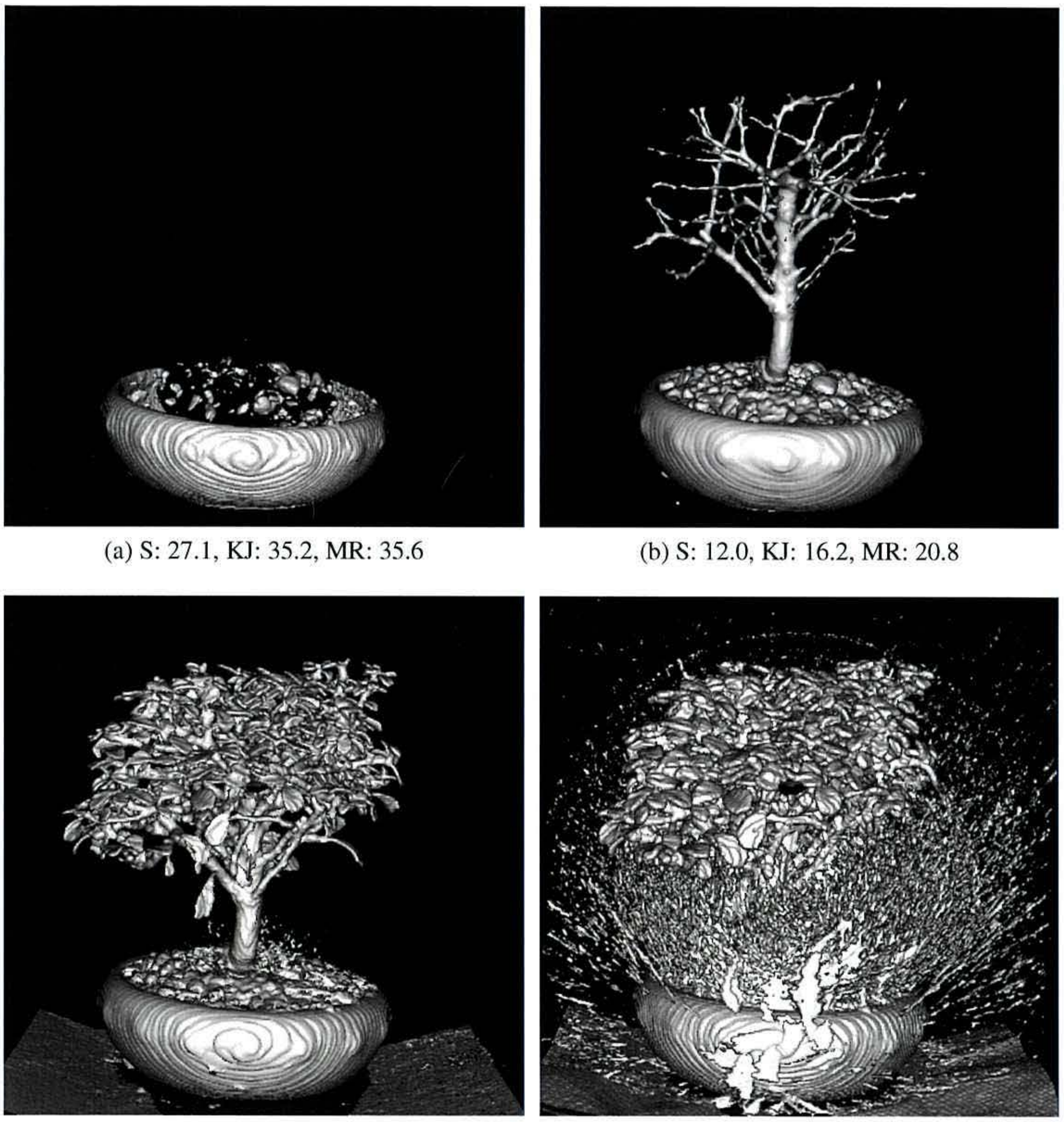
4.6 Summary

This chapter has presented Kd-Jump, a stackless traversal of implicit kd-trees for faster isosurface ray tracing. It was shown that Kd-Jump can outperform both stackless and stack-based approaches, while only needing a fraction of memory compared to a stack-based approach. Further, Kd-Jump exploits the index-based referencing used for implicit kd-trees to achieve traversal-paths equivalent to a stack-based method, without incurring the extra node visitation of kd-restart.

To further strengthen kd-tree, Hybrid Kd-Jump was introduced. Hybrid Kd-Jump utilises a volume stepper for leaf testing and a run-time depth threshold to define where kd-tree traversal stops and volume stepping occurs. By using both methods it was possible to gain the benefits of empty-space removal and hardware-based texture interpolation. It was shown that Hybrid Kd-Jump performs well at removing empty space and can outperform a brute-force ray-tracer.

Memory usage for an implicit kd-tree may be too large if min/max pairs are stored in each node. This work showed that, if the conditions for the current isosurface are moved out of traversal and into the tree nodes themselves, then significantly less memory is required. In addition, even with a naive implementation, updating the implicit kd-tree for a large volume was shown to be quite fast.

Finally this chapter showed the usefulness of loading new rays once a warp of threads completes and report that such an approach yields promising results for faster ray tracing. In addition, this chapter also discussed and examined the separation of the ray-tracing pipeline into separate kernels, and showed that the methodology has some promise for better efficiency.



(a) S: 27.1, KJ: 35.2, MR: 35.6

(b) S: 12.0, KJ: 16.2, MR: 20.8

(c) S: 7.4, KJ: 10.7, MR: 12.8

(d) S: 4.3, KJ: 6.9, MR: 7.9

Figure 4.10: Bonsai Tree rendered with a 1024^2 screen buffer, using a stack (S), Kd-Jump (KJ) and MultiRay (MR); MultiRay was based on the Kd-Jump Kernel. Full traversal of the implicit kd-tree is performed. Kd-Jump maintains a performance improvement for scenes of variable-complexity. Additionally, load balancing rays with MultiRay shows improved performance

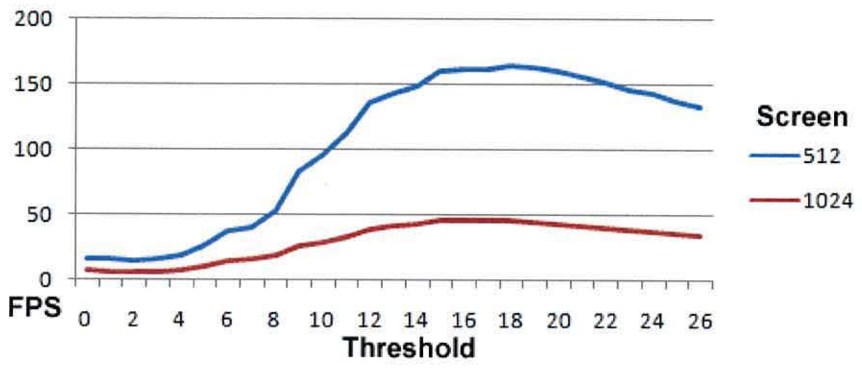


Figure 4.11: FPS for Aneurism (512^3) using Hybrid Kd-Jump (with dynamic-update) in 512^2 and 1024^2 screen resolutions, over all thresholds. Threshold of 1 is a special case where a pure volume-stepper kernel is used. The last threshold represents a complete downward traversal of the kd-tree. The graph clearly shows that the Hybrid Kd-Jump is able to remove empty space and gain in performance by reducing redundant traversal steps into deeper levels.

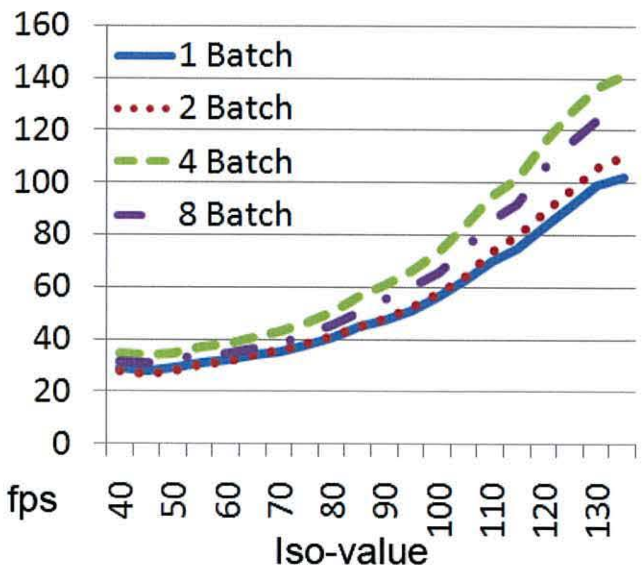


Figure 4.12: The results for loading multiple-rays per-thread once a warp terminates. Balancing workload (i.e., keeping CUDA warps as active as possible) can improve performance.

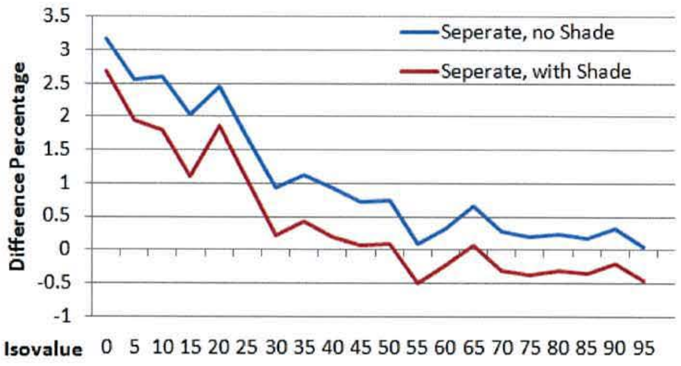


Figure 4.13: The difference in FPS (%) between separate kernels and the whole Kd-Jump kernel. For each isovalue, the FPS is averaged over different views. It shows that the separate kernels lead to only a minor improvement.

Chapter 5

Real time Semi-Automatic Volume Segmentation

Segmentation of volumetric data, whether it is medical or simulated, is the separation of a feature from the remaining space. The process of segmentation can be performed by-hand or aided by a computer. Computer aided segmentation can then be separated into two sub-topics, automatic and semi-automatic computer-aided segmentation.

A segmentation tool of volumetric data must balance between ease-of-use for the user and effectiveness of the results. While some segmentations are fully automatic, or are designed for specific anatomy or features, some users may wish to segment the data themselves. Indeed many segmentations are still required to be manually performed by an expert. A trade off is to allow a user to still direct the segmentation, but with far less input. The usefulness of being able to segment a volume, in near real time, has been shown to be useful in a clinical environment in the clinical survey by Saiviroonporn [267]. In addition the power of GPUs has been shown to improve performance of GPU-based segmentation methods, as reported by Hadwiger [115].

This chapter details a semi-automatic segmentation interface utilizing Support Vector Machines and acceleration techniques, such as incremental SVM and the utilization of a GPU for computationally intensive tasks. The hypothesis is that the training delay, after a user has stopped painting input data, can be hidden by the use of Incremental SVM, rather than be a noticeable delay if batch-training SVM was used.

5.1 Support Vector Machines

Machine learning methods, well detailed by Vapnik [305], are used to learn patterns and then predict whether new input matches those patterns; typically this involves classifying a domain based on a set of inputs. Fig. 5.1 illustrates a simple case where the domain has been divided by a line in order to separate the two classes of inputs; circles and squares. Inputs are referred to as *feature vectors*, where each vector element is an *attribute*. Thus, in Fig.5.1 there are two attributes, the x-position and y-position. Typically, applications of machine learning methods utilize feature vectors with many attributes.

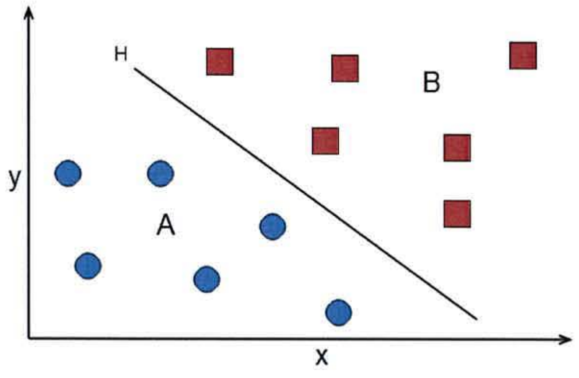


Figure 5.1: The goal of a classifier (one based on a hyperplane decision function) is to find a hyperplane which will pass in-between the class clusters.

Classifiers themselves are typically estimation functions of the form $f : \mathbb{R}^N \rightarrow \pm 1$, which are trained to a set of N-dimensional feature vectors $\mathbf{x}_i$ with corresponding class labels $y_i = \pm 1$, where $= (\mathbf{x}_1, y_1)...(\mathbf{x}_\ell, y_\ell) \in \mathbb{R}^N \times \pm 1$ [127]. A function is chosen such that when given a new example $(\mathbf{x}, y)$,

$$f(\mathbf{x}) = y.$$

A good set of learning functions [45] for f, which Support Vector Machines are based on [127], are the class of hyperplanes

$$(\mathbf{w} \cdot \Phi(\mathbf{x})) + b = 0 \quad \mathbf{w} \in \mathbb{R}^N, b \in R$$

with corresponding decision functions

$$f(\mathbf{x}) = \mathrm{sign}((\mathbf{w} \cdot \Phi(\mathbf{x})) + b)$$

As illustrated in Fig. 5.1, the data can be linearly classified by a simple line. However, in a general case it is not the situation that a linear line, plane or hyperplane can separate the two classes in the input space; for example Fig. 5.2(a). To do this, nonlinear classifiers are needed.

To solve the problem of finding a nonlinear hyperplane by which to separate (for classification) two classes, the input vectors can (theoretically) be nonlinearly mapped to a *feature-space* using Φ. It is then possible to find a *linear optimal hyperplane* in the feature-space, which will then correspond to a nonlinear decision function in the input-space; as illustrated in Fig. 5.2. The novelty of SVMs is that the map to feature-space need not be computed directly and is implicitly defined by the use of kernels. Specifically, Boser [34] showed that a hyperplane can be computed while working in

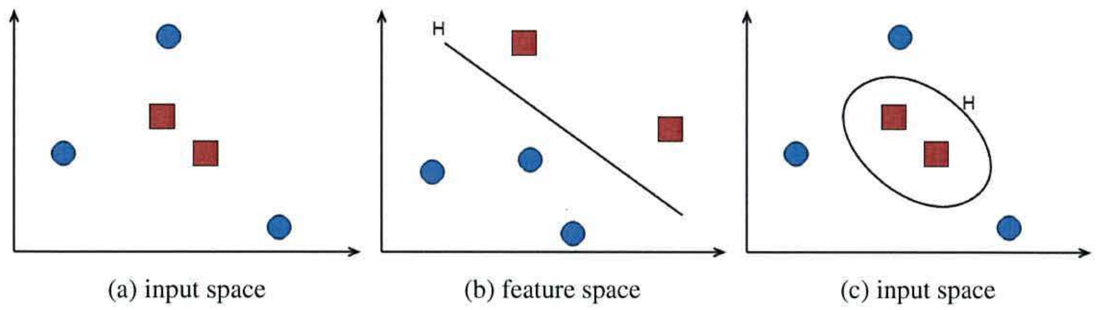


Figure 5.2: Forming a nonlinear SVM classifier is achieved by implicitly mapping the input space (a) to the higher-dimensional feature-space (b) and finding a linear optimal hyperplane there. The resulting hyperplane results in nonlinear classification back in input space (c)

the kernel dot-product space (so long as it is positive-definite) and that mapping of the input vectors into higher dimensional-space is implicitly achieved [127]. Thus, the map Φ can be replaced with a kernel to define standard SVM decision function for classification:

$$f(\mathbf{x}) = \text{sign}(\sum_{i}^{\ell} v_i \cdot k(\mathbf{x}, \mathbf{x_i}) + b)$$

where k is the kernel function. A common kernel for pattern recognition, and used in this thesis, is the RBF kernel:

$$k(\mathbf{x}, \mathbf{y}) = \exp(-\|x-y\|^2/2\gamma^2)$$

The hyperplane is constructed such that it maximizes the margin between the convex hulls of the two input classes, as detailed by Hearst [127]. It is typically computed by solving a constrained quadratic programming problem, the solution of which will be $\mathbf{w}$, with the expansion

$$\mathbf{w} = \sum_{i} v_i x_i$$

where input vectors with non-zero coefficients v_i are a subset of the original training data called support vectors. Alternatively, methods using Least-Squares in SVM classifiers solve the problem by finding a solution for a set of linear equations, as detailed by Suykens and Vendewalle [293].

Support vectors lie on the margin hyperplane and contain all the information needed to reproduce the problem during prediction. The remaining input-vectors (those that are not support vectors) will not contribute to the decision function when predicting the class of new examples.

Early SVM methods were based on a hard-margin hyperplane, where a hyperplane is found for all input-vectors. A common problem is how to find an optimal hyperplane if there are errors or noise in the training data. The solution is to use the soft-margin approach detailed by Cortes [71], where an error function and a regularization parameter C are introduced. The attribute C can be viewed as the cost of fitting the hyperplane to inputs with high error. Those inputs with high error will generally not influence the hyperplane, as illustrated in Fig. 5.3.

To summarize, standard Support Vector Machines are learning functions that are trained once with set of labeled inputs, in order to find a separating hyperplane, which maximize the margin between two classes, such that when a new example is given the SVM decision function will estimate its class label.

5.1.1 Incremental SVM

Support vector machines, as detailed by Vapnik [305], require training before they can be used to predict the class of new examples. If all input data is known then there is no problem (in the context of this thesis) as the SVM model can be trained once and then used. However, if the input is not known *a priori* and/or is altered then this

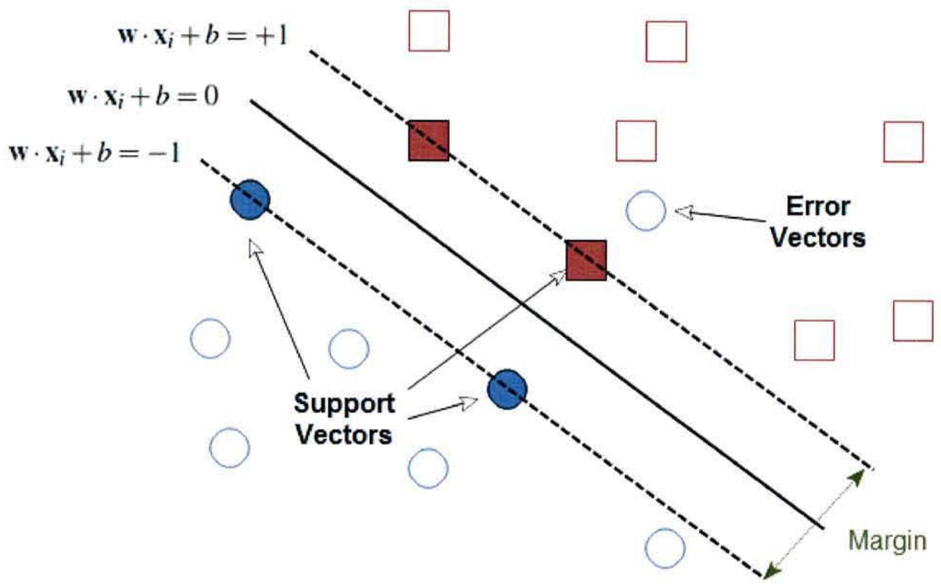

$\mathbf{w} \cdot \mathbf{x}_i + b = +1$
$\mathbf{w} \cdot \mathbf{x}_i + b = 0$
$\mathbf{w} \cdot \mathbf{x}_i + b = -1$
Error Vectors
Support Vectors
Margin

Figure 5.3: SVM with soft-margins. A hyperplane that maximizes the margin between input classes is found, but inputs too costly to fit are (effectively) ignored as errors. Input-vectors on the margin are support vectors as they limit the margin width. Others input vectors not on the margin will have no impact in the standard SVM decision function.

would require re-training the SVM model again, with *all* input vectors. In the context of a segmentation tool, where the user is likely to make mistakes and alter their input, constant re-training could easily result in a notable delay whenever the SVM model is required for classification. This would be especially so if the number of input vectors ranges in the thousands.

Incremental SVM, also known as Online SVM is a way to train very large datasets one example at a time. Initially introduced by Syed [295], it was not until the work by Cauwenberghs [55] that an accurate method for incremental and decremental SVM learning was developed. Due to the nature of learning one example at a time, additional input vectors can be added very quickly. Also of high importance, especially for interactive segmentation, is the ability to *unlearn* training examples. Decremental learning is a method to accurately remove (or unlearn) one example from an already trained SVM model.

The method by Cauwenberghs [55, 80] (derivations partially reproduced for completeness in this thesis) employs an SVM of the form $f(\mathbf{x}) = \sum_{i=1}^{N} y_i \alpha_i \cdot k(\mathbf{x}_i, \mathbf{x}) + b$, which is trained to the inputs $(\mathbf{x}_i, y_i) \in \mathbb{R}^m \times \pm 1 \forall i \in 1, \ldots, N$ by solving the dual form quadratic program

$$\min_{0 \leq a_i \leq C} W = \frac{1}{2} \sum_{i,j=1}^{N} a_i Q_{ij} a_j - \sum_{i=1}^{N} a_i + b \sum_{i=1}^{N} y_i a_i \tag{5.1}$$

with Lagrange multiplier (and offset) b and $Q_{ij} = y_i y_j k(\mathbf{x}_i, \mathbf{x}_j)$. The Kuhn-Tucker [172] conditions uniquely define the solution of the dual parameters $\{\alpha, b\}$ that minimize the above problem:

$$\begin{aligned} g_i &= \frac{\partial W}{\partial \alpha_i} = \sum_{j=1}^{N} Q_{ij} \alpha_j + y_i b - 1 \begin{cases} > 0 & \alpha_i = 0 \\ = 0 & 0 \leq \alpha_i \leq C \\ < & 0 \alpha_i = C \end{cases} \\ h &= \frac{\partial W}{\partial b} = \sum_{j=1}^{N} y_i \alpha_j \equiv 0 \end{aligned}$$

Cauwenberghs [55] groups trained input-vectors into three sets; margin-vectors; error-vectors; and reserve-vectors, where the value g_i determines which set the training input belongs to. With $g_i > 0$ the input lies outside the margin and is added to a set of reserve vectors that do not influence the decision function. If $g_i = 0$ then the input-vector is on the margin hyperplane, is a support vector and is added to the margin-vector group. Finally, if $g_i < 0$ then the input is within the margin, or classified incorrectly, and is added to the error-vector group. Fig.5.4 illustrated a basic classification of feature space using Incremental SVM using a soft-margin hyperplane.

Due to the nature of Incremental learning all input-vectors are maintained and can be moved into other sets during the course of training. By maintaining the Kuhn-Tucker conditions for all inputs, incremental SVM can successfully update a trained model to incorporate new training examples – the derivations for Incremental SVM learning are beyond the scope of this thesis; please see the work by Diehl and Cauwenberghs [80].

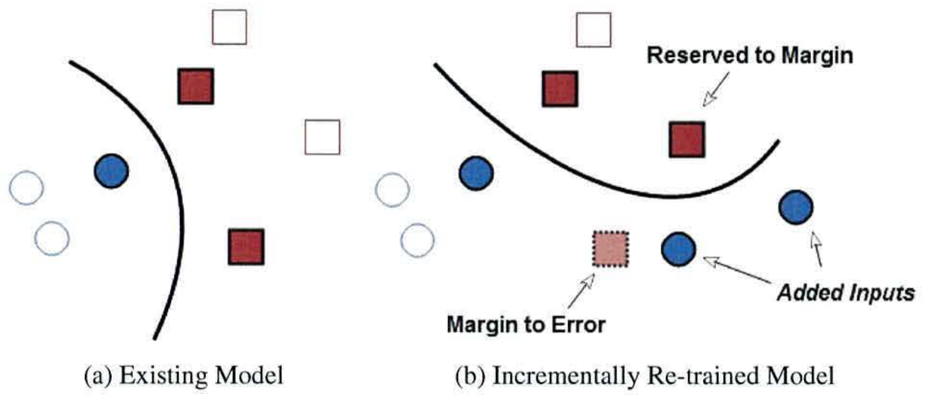


Figure 5.4: Incremental SVM updates an already trained model by learning new examples and updating previous examples. For example, the new inputs added in (b) change a previously learned input in (a) from being a margin-vector to an error-vector, while a previous reserve-vector becomes a margin-vector

5.2 Fast Segmentation Using Incremental SVM and CUDA

Segmentation, in its most basic form, is the process of defining a region within a volume. This region can be classed as the wanted region, while the remaining space can be classed as the unwanted region. As detailed in the previous section, an SVM can be used to classify a domain into two classes. This thesis introduces a tool based on incremental SVM, where training inputs are provided by a user. To provide training data, the user paints onto a two-dimensional plane, which is aligned to one of the volume dimensions and displays the volume data it passes through. Fig. 5.7 shows the introduced tool, where training inputs (shown as red and green) have been painted onto the input-plane.

The segmentation software is divided into several distinct parts; user input, training, classification and visualization. All parts must be as fast as possible and to this end various methods and techniques can be employed. Specifically, there are low-level software-enhancements and high-level algorithm-enhancements utilized. The primary means by which the software is accelerated is with the incorporation of CUDA, multi-CPUs, OpenGL and Incremental SVM. The segmentation software utilizes the incremental SVM implementation described by Diehl and Cauwenberghs [80], which is available as Matlab code[79].

Motivation for the work was originally to provide a simplified version of an existing segmentation tool, presented previously by Tzeng [301, 300], such that school children could use it to interactively learn about anatomy. Children were able to segment anatomy from a variety of data-sets. The best results were then sent to a 3D-printing company to be made into solid objects, as shown in Fig. 5.5.

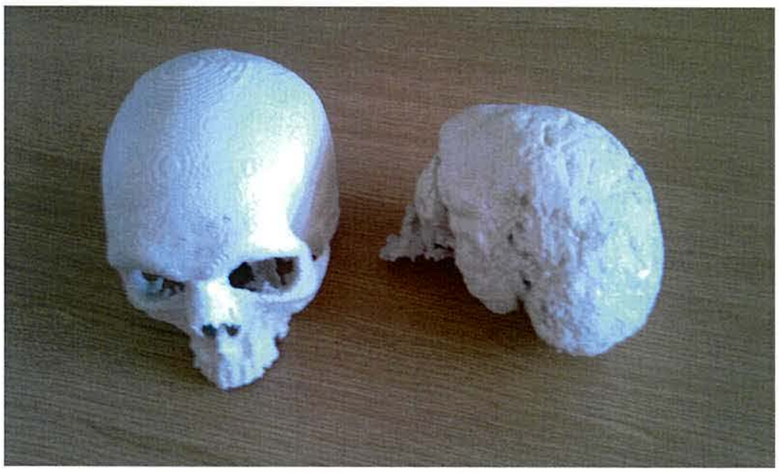


Figure 5.5: Photo of 3D-printed objects based on segmentation data

Originally, the software developed (that later became the basis for this work) for interactive learning of anatomy used a batch SVM software library called LibSVM [57]. In addition, the whole volume classification was performed using LibSVM, which typically required approximately 50-60 seconds to calculate. In order to reduce this time, the developed software was made to incorporate a GPU-based SVM decision function. Rather than use LibSVM, the new CUDA-based code predicted the class of each volume voxel. Originally, the classified volume would be downloaded from the GPU and used in a CPU-based volume renderer. Finally, due to noticeable delays to train and retrain SVM models during segmentation, especially when difficult problems arose, incremental SVM was incorporated. The culmination of these improvements led to the new software implementation described in the following sections.

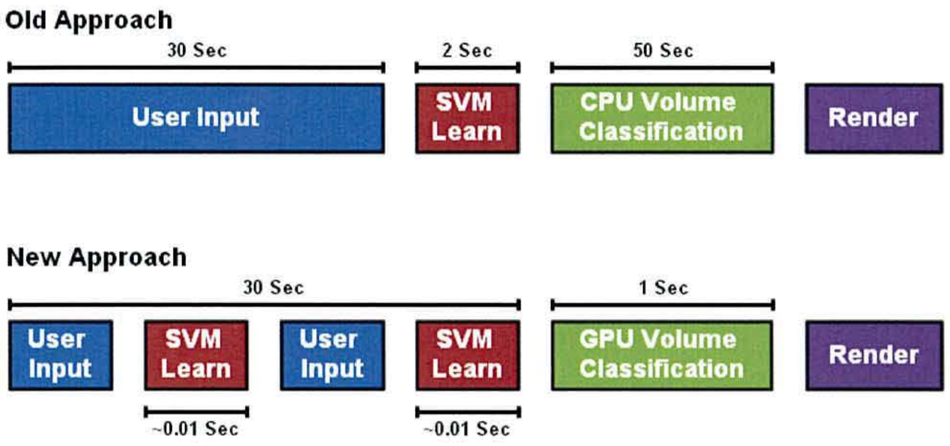


Figure 5.6: Comparison of previous approach (top) against the new approach (bottom). With the new approach there is no noticeable delay because of SVM training, once the user has finished supplying input. The delay waiting for volume classification is also significantly reduced by utilizing GPU's

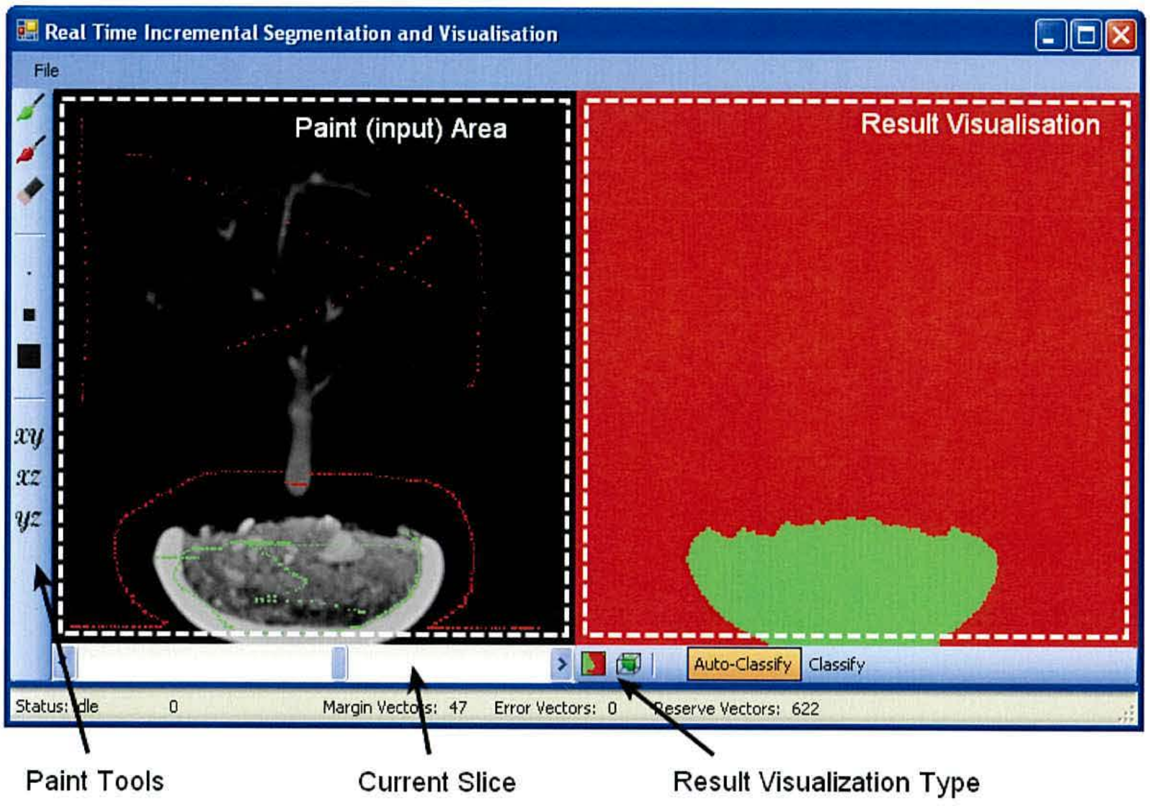
In the work by Tzeng [300] a novel interface was presented which allowed a user to paint on a 2D volume slice. Once painting was complete, the software would convert the painted inputs into training data, train a Neural Network and then apply the trained model to predict the full volume class values. In later work, Tzeng [301] replaced the Neural Network learner with an SVM. In addition, a GPU-based volume classifier and renderer was introduced for the Neural Network method, but not for the SVM method; due of its complexity. The Neural Network shader was able to classify and render a 256^3 volume in under 0.75 seconds. Training of the Neural Network was, however, reportedly much slower than with the SVM trainer.

5.2.1 Integration of CUDA

CUDA [225] is a GPU programming architecture for GPUs. The segmentation software utilizes the GPU resources and computation-power throughout to enhance the performance. Specifically, the volume data is stored directly in the GPU memory rather than in the system memory. The main reason for this design decision is that most of the segmentation software, i.e. those functions which require access to the volume, can be accelerated by CUDA kernels; whether they be parallel tasks or simply one-thread tasks. For example, generation of the input-vectors is done on the GPU directly and passed to the SVM trainer.

5.2.2 GUI Overview

The segmentation software GUI developed for this thesis, as seen in Fig.5.7, has functionality similar to the software presented by Tzeng [300]. Specifically, an area for the user to paint training data, an ability to alter the depth of the slice on the current view plane, and an ability to observe the result. Results are visualized as either a classification plane or as a direct-volume rendering with only the segmentation regions visible.



Status: Idle 0 Margin Vectors: 47 Error Vectors: 0 Reserve Vectors: 622

Figure 5.7: Screen capture of the segmentation GUI in use

User input is required to produce the necessary training data for the SVM. User input is acquired using a two-dimensional slice-plane and paint-brush tools, where the view shows a single slice of the original volume, at a set depth. The depth can be adjusted by the user using a slide control, which is situated below the viewing area. The user selects a paint brush from the toolbar, selects a depth within the volume along one of three viewing axis and then clicks on the slice-plane to begin painting.

The user is required to paint two groups, a region of green-paint that is the desired object or region, and a second region of red-paint, which typically encompasses the green-paint in order to limit the region size; as depicted in Fig. 5.11. If the segmentation is incorrect, then the user must erase paint information from the areas the user has determined are incorrect and repaint new inputs. Alternatively, if a segmentation is to be refined, the user may simply add new paint information without needing to override previous input. An example may be when a user is segmenting a complex object through 3D-space. In such a case they may be required to pick a new depth within the volume to add new paint information, such that the system correctly segments the object as it changes shape through the volume.

5.2.3 Handling User Input

Internally, once paint-events occur, the software tool, developed for this thesis, first checks the existing paint information for the voxel(s) the user specified. If no paint data already exists for the voxel then a training-input event is triggered. There are several features of the input painting which could see a slowdown if implemented naively. Firstly, once it is determined the paint information either adds or alters input to the SVM, the SVM must be updated accordingly. If this process was performed with a single system-thread (and subsequently within a single CPU) then there would be a noticeable minor slowdown in the speed with which the user is able to paint, as the SVM incrementally learns the input.

To ensure the computational aspects of adding or altering an input point does not hinder the speed of the system, as perceived by the user, we can incorporate multiple threads. The first thread is tasked with running the GUI, the basic handling of the events as they occur and low-level OpenGL render updates. The second thread is charged with managing the queue of paint events, testing for input changes and updating the SVM with incremental changes.

With multiple threads the user is free to paint as much input as they wish without any noticeable delay (during painting). As a paint event occurs (mouse click and mouse move) the paint location is stored within a paint event queue. This queue is shared between the primary and secondary threads and locked whenever either access it. As the information (paint location and event type) is relatively small the queue need only be locked for a small duration as the threads update it. Once the data has been inserted/removed the threads are free to continue operating and the queue can be unlocked ready for further access by any thread. This is especially important as the secondary thread is tasked with retrieving paint events and then performing a large amount of computation to update the SVM. In a single-thread scenario, if the user had to wait for the SVM to be updated each time they clicked to add paint information there would be significant delays noticeable; especially if the SVM has difficulty finding a solution for a particular input.

5.2.4 Generation of Training input

The training input is comprised of several features that make up the multi-dimensional input-vector for the SVM utilized in this thesis software. As illustrated by Fig.5.8, these are:

- **Position** - the three-dimensional cartesian-axis position of the training point within the volume, given as normalised X, Y and Z. Makes up the first 3 dimensions of the training vector.
- **Value** - the normalized volume-value of the training point. Makes up the 4th dimension of the training vector
- **Neighbour Value** - the normalised volume-values for the immediate 6-point neighbors of training point. Makes up dimensions 5 to 11.

- **Gradient Magnitude** - the un-normalized first-derivative at the input point. Derived from the normalised volume, using central-difference. Makes up dimensions 12 to 14.

The volume, stored on the GPU, is bound to a texture reference. This enables cached-access to the data (using the texture-access functions), which is specifically useful if access is localized in a specific region. In the case of producing the training vector, there are several localized memory accesses, which will utilize the texture-caching feature.

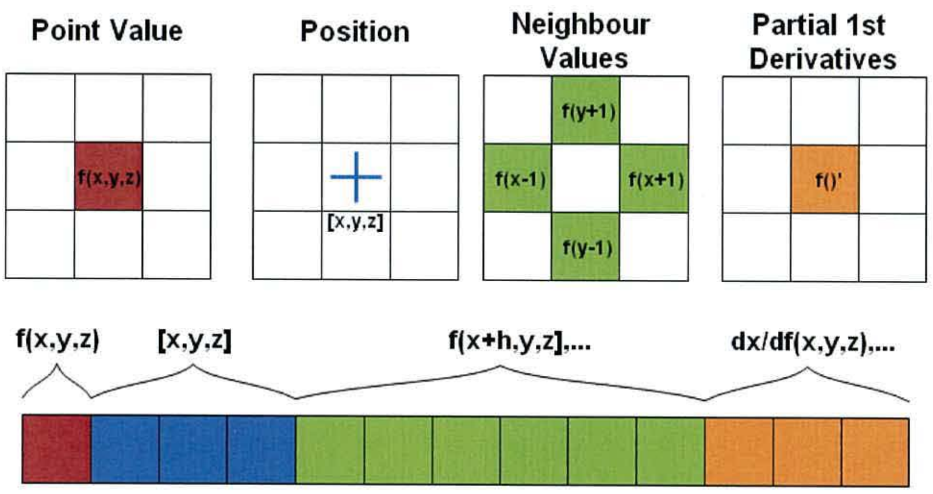


Figure 5.8: Illustration of an input vector as used for segmentation

Creating the training vector is performed by a specialized kernel, which is initialized with the training point and a pointer to the output vector for storage. The kernel is programmed to set the vector with the relevant data; as listed previously. The position dimensions of the training vector are immediately stored and the volume value at this position is also accessed and stored. The 6-neighbour volume-values are then accessed using offsets to the original training point. By setting the texture reference to clip texture-accesses, no additional code is required to test for out-of-bounds cases.

Finally, with the neighbour values accessed and stored, the gradient vector is computed with central-difference.

5.2.5 Visualization Backbone

There are several visualizations occurring in the segmentation software; either being two-dimensional or three-dimensional. The basic approach for incorporating these visualizations in the software has been to utilize the Tao framework [1]. The Tao framework allows for OpenGL controls to be incorporated into the .Net software UI. Not only does this ensure fast rendering, but vastly simplifies the code by allowing OpenGL to automatically manage the screen rendering and view resizing. The other reason for utilizing OpenGL is to allow CUDA inter-operability. By supplying the OpenGL context to CUDA, CUDA can directly bind to a Pixel Buffer Object, which then allows CUDA kernels to directly render to the screen output. The user can specify the render mode for the output screen as either being a 2D texture showing the classification results for the currently-selected volume slice or the DVR visualization of the classified volume.

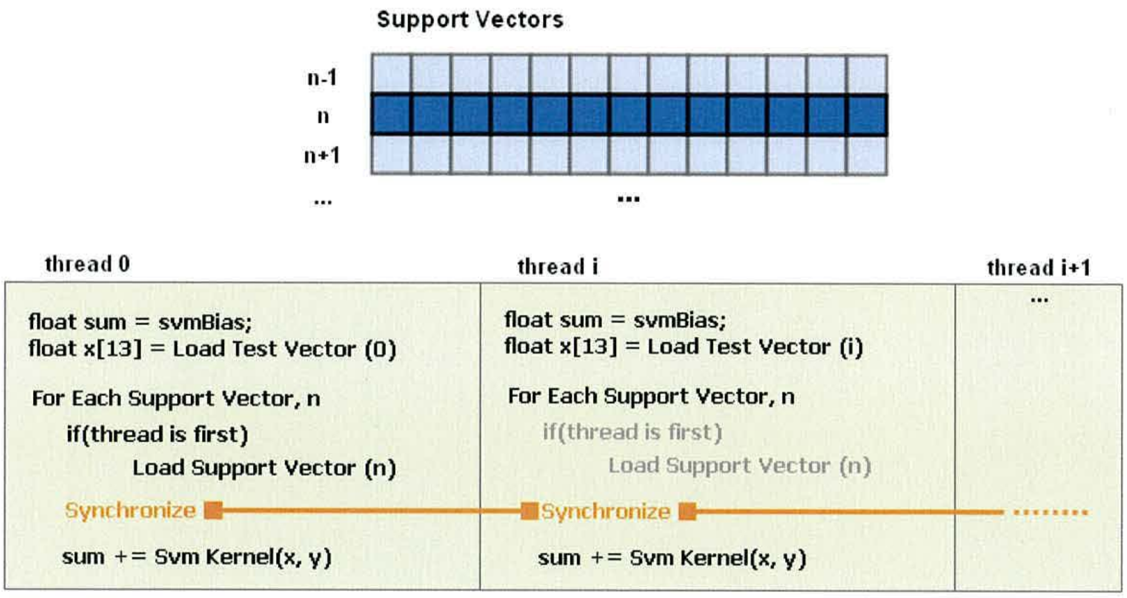
5.3 Volume Classification

Once an SVM has been trained it can be used to classify the remaining volume into two-classes; wanted and unwanted. The process of classifying the volume is completely parallel and does not require conditional branching at all; thus is ideal for GPU computation. To classify the volume there are several steps which must be accomplished. Firstly, the relevant SVM information, such as support vectors and SVM parameters, must be uploaded to the GPU. Secondly, each voxel of the volume must be converted to an N-dimensional input-vector. Finally, the SVM is evaluated for each voxel vector and the predicted class stored.

Volume classification is separated into two kernels, generation of the test vectors and SVM evaluation. These two tasks are separated in order to maximize the efficiency of each kernel. As previously detailed, it is favorable to utilize light-weight kernels as results in maximum occupancy of the GPU multiprocessors.

Conversion of each voxel into a test vector is quite simple and nearly identical to the generation of training input (as described in section 5.2.4). However, the kernel for volume classification must produce a test vector for each voxel. Due to limited resources, only a finite number of vectors can be tested at any given moment. As a result, computation is divided into several batches, where only a small sub-set of the volume is tested at a time. The batch size is rather arbitrary and completely determined by the amount of available memory on the GPU. However, for this work a batch size of 256×1024 (number-of-threads $\times$ number-of-blocks) was chosen. With the 13-dimensional feature space utilized, this approximately requires 13MB of GPU memory to store the test vectors. Once the test vectors have been created and stored, they are passed to the SVM evaluation kernel.

The SVM evaluation kernel is designed such that each thread is responsible for evaluating a single test vector, as well as storing the predicted class for the corresponding voxel. Due to the limited resources of CUDA kernels, and the fact SVM support vectors require a considerable amount of memory, SVM evaluation must be implemented with care. Specifically, the kernel is designed such that each support vector is loaded once into shared memory, as shown in Fig. 5.9. By transferring one support vector at a time into shared memory and forcing all threads to operate synchronously with the currently loaded support vector, redundant access to global memory can be



```
float sum = svmBias;
float x[13] = Load Test Vector (0)

For Each Support Vector, n
    if(thread is first)
        Load Support Vector (n)
    Synchronize
    sum += Svm Kernel(x, y)
```

```
float sum = svmBias;
float x[13] = Load Test Vector (i)

For Each Support Vector, n
    if(thread is first)
        Load Support Vector (n)
    Synchronize
    sum += Svm Kernel(x, y)
```

Figure 5.9: Basic overview of SVM evaluation CUDA-kernel. All threads load their own test vector, while only the first thread loads support vectors into shared memory. Threads must wait (at the synchronization point) for the first thread to complete loading the support vector.

avoided and the maximum efficiency can be attained.

5.4 Segmentation Results

For all testing the SVM was set to use a Gaussian RBF kernel with a scale (gamma) of 0.1 and C set to 1000. All values for all dimensions in the feature space were normalized between 0 and 1. For volume prediction, the timing results were recorded using CUDA's high resolution timers. Any delays observed during test, after the final input was painted, are given as approximations.

A test was performed which resulted in many support vectors being required to fit the hyperplane, the result of which is given in Fig.5.11. This represents a complex case due to the locality of the segmentation in only 2 of the 13 feature dimensions, which ultimately results in many support vectors. In contrast, generally, segmentation of objects from volume data will not result in as many support vectors. The results show that with approximately 500 support vectors, prediction of the entire 256^3 volume was accomplished in 2.12 seconds, while training of the final inputs (after painting had ceased) required approximately 0.25 seconds.

The results for segmentation tests are shown in Table 5.1 with the corresponding screen captures shown in Fig.5.10. Interestly, when segmenting a specific object, the number of support vectors is much lower than in the previous test case. As a result a noticeably smaller time (less than a second) was needed for the full volume prediction.

Table 5.1: Statistics for results shown in Fig.5.10.

	Prediction		**SVM vectors**		
	size	time	margin	error	reserve
Bonsai basin	256^3	0.62 (sec)	159	0	2632
Engine Cap	$256^2 \times 128$	0.66 (sec)	163	0	1230
Skull side	256^3	0.58 (sec)	137	0	3000

5.5 Discussion

Using incremental SVM, rather than a standard non-incremental SVM, has advantages and disadvantages. With a normal SVM, where the input is gathered first and then sent for training at the same time, the GUI does not have to manage the SVM. Specifically, with incremental SVM, care has to be given to ensure that if input needs to be removed or changed, then the correct vector is unlearned. While this does require additional work and maintenance, an incremental SVM is much faster than a standard SVM. This is especially so for the user, as assuming paint operations are not delayed when adding new input to the SVM, the user is oblivious to the fact the SVM is being trained.

It is questionable whether using 13 features for segmentation is useful. The initial hypothesis would be that with more features present in the input-vectors, that an SVM would more easily predict the volume classes. However, this was not the observed case during testing, contrary to the discussion given by Tzeng [301].

Segmentation typically entails the separation of an object or group of objects from the remaining space. These objects are easily distinguished by their location and intensity. Thus, including the position and volume intensity, as features, is of the greatest importance. However, whether including the neighbor values or local normal is useful is not clear. What is certainly clear is the problem these features can cause. Specifically, what was observed was small areas outside the desired spatial area being predicted as *desired*.

In most cases, the incorrectly predicted areas corresponded to edges or areas with high gradient. Subsequently, one could deduce that the SVM was improperly trained by the user, and that local gradient and/or local neighbor features caused undesired predictions outside (spatially) the wanted object.

5.6 Context-Preserving Rendering

A segmentation-volume defines unique regions within a volume of intensities (CT or MRI data for example). Typically the segmentation-volume is used to customize the rendering of the intensity-volume. However, it is possible to directly render the segmentation-volume without considering the intensity-volume as was shown in the work by Hadwiger [114].

This section details a context-preserving visualization method for segmentation data, utilizing curvature information. Whilst this method is not incorporated into the segmentation GUI described in this chapter, it may prove useful in future work; see Section 5.7.

A standard DVR approach is to project rays into the volume and compute the discreet approximation of the DVR integral using the front-to-back formulation of the over-operator [249, 37]. Given a point along a ray $\mathbf{P}$, an integer volume can be accessed $f(\mathbf{P})$ to obtain a segmentation ID. Using front-to-back composition, it is possible to compute the opacity α_i and colour c_i contribution of each sample at each step along the ray:

$$\begin{aligned} \alpha_i &= \alpha_{i-1} + \alpha(\mathbf{P}_i) \cdot (1 - \alpha_{i-1}) \\ c_i &= c_{i-1} + c(\mathbf{P}_i) \cdot \alpha(\mathbf{P}_i) \cdot (1 - \alpha_{i-1}), \end{aligned} \tag{5.2}$$

where $\alpha(\mathbf{P}_i)$ and $c(\mathbf{P}_i)$ is the opacity and colour contributions at $\mathbf{P}_i$, and where α_{i-1} and c_{i-1} are the previously accumulated opacity and colour values along the view ray.

5.6.1 Curvature-Based Context Preservation

Rendering the segmentation volume using purely DVR would result in large regions appearing largely opaque. Even with the use of transfer functions it would be hard to simultaneously remove regions obscure interesting features and preserve the important visual cues of those regions. In the approach described below the idea is to make areas with little or no curvature transparent, while having features with high curvature more opaque; for segmentation data only. It is also interesting to consider the distance of the sample point from the camera, such that samples closer to the view are more transparent. Additionally this work also considers the previously accumulated opacity along each ray such that it will reduce the contribution of subsequent samples if a high amount of opacity has already been accumulated.

In order to incorporate curvature and surface angle quantities into the proposed composition scheme, a volumetric scalar-field of normalized curvature values is defined as $\mathbf{k}(\mathbf{P}_i) \in [0..1]$, where $\mathbf{k}_{\mathbf{P}_i}$ is the mean curvature value at $\mathbf{P}_i$. Also defined is the normalized gradient of a sample $\hat{\mathbf{g}}_{\mathbf{P}_i}$.

The combination of the curvature, distance, accumulated opacity and angle quantities leads to the following new opacity equation for all samples $\mathbf{P}_i$:

$$\alpha(\mathbf{P}_i) = \alpha_{tf}(f_{\mathbf{P}_i}) \cdot m(\mathbf{P}_i), \tag{5.3}$$

with

$$m(\mathbf{P}_i) = (\mathbf{k}_{\mathbf{P}_i})^{\left(k_t \cdot (1-\alpha_{i-1}) \cdot (1-d) \cdot ||\hat{\mathbf{V}} \cdot \hat{\mathbf{g}}_{\mathbf{P}_i}||^{ke} \cdot (1-(\mathbf{k}_{\mathbf{P}_i}))\right)^{ks}}, \tag{5.4}$$

where $d = ||\mathbf{P}_i - E||$ is the normalised distance between the sample point and the camera E, is α_{i-1} is the previously accumulated opacity, $||\hat{\mathbf{V}} \cdot \hat{\mathbf{g}}_{\mathbf{P}_i}||$ is the absolute of the dot product between the surface normal and the view direction and k_e controls view angle effect. The $1-(\mathbf{k}_{\mathbf{P}_i})_{[0..1]}$ term factors in the curvature for a second time and has the effect of ensuring lower opacity for samples with low curvature. The term $||\mathbf{P}_i - E||_{[0..1]}$ factors in the distance of the sample at $\mathbf{P}_i$ from the view location E. The distance quantity is also normalised to the range $[0..1]$ such that $||\mathbf{P_0} - E|| = 0$ represents a point at the view location and $||\mathbf{P_N} - E|| = 1$ represents the sample point furthest away from the camera. The term $||\hat{\mathbf{V}} \cdot \hat{\mathbf{g}}_{\mathbf{P}_i}||$ has the effect of ensuring surfaces viewed side-on to the camera will be more opaque and gives important view-dependent edge cues. Finally, the term $1-\alpha_{i-1}$ factors the previously accumulated opacity such that the opacity contribution of the current sample is reduced.

There are two adjustable parameters available in Eq.5.4, k_s and k_t. k_t controls the the amount of opacity reduction and relates to the distance factor. With high values for k_t the opacity of samples closer to the camera are reduced, as shown in Figure 2. The k_s parameter controls the sharpness of transition between high and low opacity contribution. A low value for k_s will result in a smooth transition while a high value for k_s will result in a sudden transition between low and high opacity, as shown in Figure 2.

5.6.2 Focus Region

Being able to specify a location of interest and rendering that area differently can be important for users. Our approach is derived from the work by Kruger [169] who chose between the unit distance and curvature value of a sample point to modulate the opacity contribution of a sample.

We define a region of interest by defining a focus point $\mathbf{S}$ and a focus radius h. The opacity of a sample using a focus region is:

$$\alpha(\mathbf{P}_i) = \begin{cases} \alpha_{tf}(f_{\mathbf{P}_i}) \cdot t^{k_d} & \text{if } t^{k_d} > m(\mathbf{P}_i) \\ \alpha_{tf}(f_{\mathbf{P}_i}) \cdot m(\mathbf{P}_i) & \text{otherwise} \end{cases} \tag{5.5}$$

with

$$t = clamp\left(\frac{||S - P_i||}{h}\right) \tag{5.6}$$

where $clamp\left(\frac{||S-P_i||}{h}\right)$ computes the unit distance of the sample to the focus point. All sample points outside the radius are clamped to one 1 using the *clamp* function. The parameter k_d controls the transition between the low and high opacity within the region of interest. The opacity modulation defined in Eq.5.5 has the effect of reducing opacity in a localized region such that if the curvature based modulation if greater than t then it is chosen instead, as shown in Figure 5.12(a). An obvious derivation of the equation is to invert the effect of the distance such that the focus point is opaque, as shown in Figure 5.12(b):

$$t = 1 - clamp\left(\frac{||S - P_i||}{h}\right), \tag{5.7}$$

however this rendering on may not be useful on its own. An additional method is a combination of both effects to give a supporting perception cue for a user:

$$\alpha(\mathbf{P}_i) = \begin{cases} \alpha_{tf}(f_{\mathbf{P}_i}) \cdot \beta & \text{if } \beta > m(\mathbf{P}_i) \\ \alpha_{tf}(f_{\mathbf{P}_i}) \cdot m(\mathbf{P}_i) & \text{otherwise} \end{cases} \tag{5.8}$$

with

$$\beta = \begin{cases} 1 - clamp\left(\frac{||\mathbf{S}-\mathbf{P}_i||-h}{h}\right) & \text{if } t = 1 \\ t^{k_d} & \text{otherwise} \end{cases} \tag{5.9}$$

where t is defined in Eq.5.6. The effect is to have the focus point and the area outside double the focus radius rendered using the feature modulation, while the area inside double the focus radius is rendered using the distance modulation if it is greater than the feature-based modulation. The result of Eq.5.9 is shown in Figure 5.12(c).

5.6.3 Implementation and Results

The work implemented the context-preserving rendering purely in software and on CPU. The normals and curvature at the segmentation object boundaries were pre-computed. We store the normals and curvature only for segment boundaries by utilising an index volume. The curvature and normals for each boundary position were pre-calculated and stored prior to rendering. For each boundary position we examined its neighbor boundary points (of the same segment) and extracted them as spacial points. We then apply principle component analysis on the found points and calculate the curvature and normal, for the current boundary point, from the eigen vectors and values.

The context-preserving rendering was tested using an altered version of the segmented Visible Human head. The size of the head volume is $573 \times 330 \times 220$ and the results are shown in Fig. 1 and 2.

5.7 Summary

This chapter has presented a semi-automatic segmentation tool accelerated by GPUs. The segmentation GUI allows a user to paint wanted and unwanted markers directly onto the volume. These markers then correspond to input data for a support vector machine. Once trained the SVM is utilized to predict the class of each voxel in the volume, which in turn can be visualized with DVR such that only the segmented object is visible.

The main contribution of this work can be summarized as being;

- The use of incremental SVM to hide the training process while the user supplies the input.
- The addition of vector labels to simplify the process of finding and unlearning trained vectors.
- The use of CUDA to accelerate the generation of input vectors.
- The use of CUDA to accelerate the class prediction for each volume voxel.

The results showed that by using Incremental SVM, no noticeable delays where observed after painting of training data. However, while the GPU-based volume classifier was able to compute the class-predictions for an entire 256^3 volume in under a second, for the general cases, it is not ideal for a truly interactive segmentation. If a user wished to see a how a segmentation developed, by viewing the whole volume prediction, as they painting inputs, then a new system would be required. Specifically, SVMs divide the space with a hyperplane that maximizes the margin between class clusters. As such we need only attempt to predict the class of volume voxels within the SVM margin, as only within this area are we to find the hyperplane. Outside of the SVM Margin a system could simply flood-fill the areas after the boundary between the classes has been found. With such a system, the number of voxel predictions needed could be dramatically reduced, so much so as potentially allow real-time volume prediction and rendering as the user incrementally updates the SVM model.

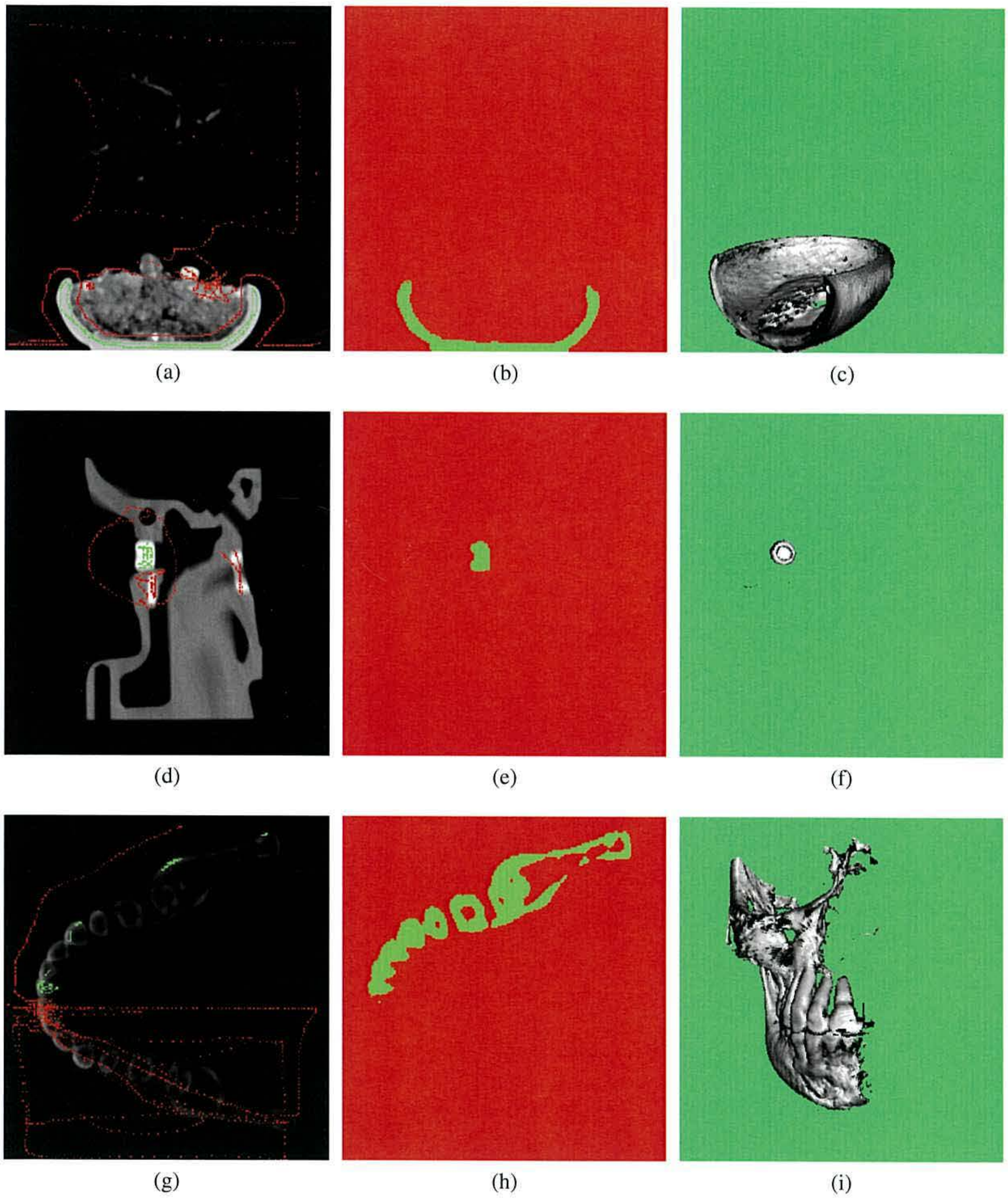
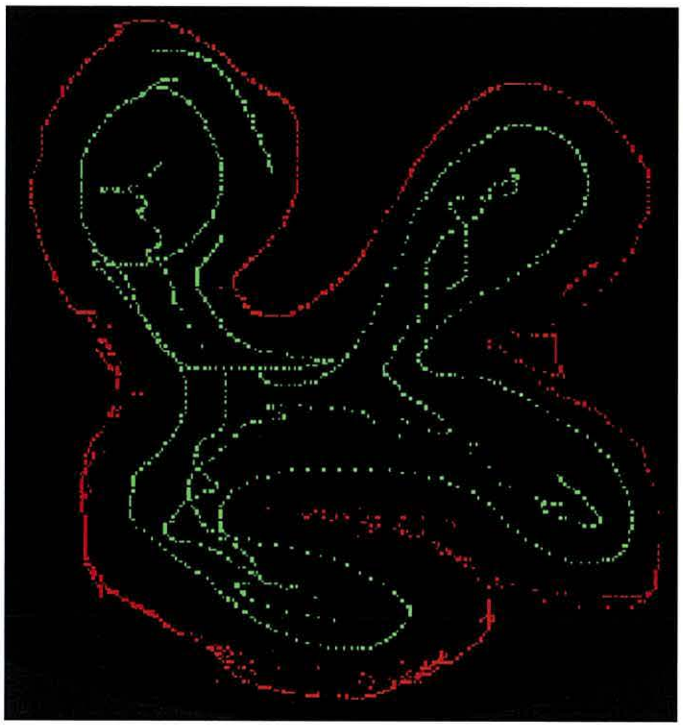


Figure 5.10: The left column show the user input data on one volume slice. The middle column shows the predicted class on the same slice, while the right column shows the full volume prediction with volume rendering. Images (a,b,c) show the segmentation of the basin from the bonsai data-set. In images (d,e,f) a portion of the engine data-set has been segmented, while in images (g,h,i) half of the skull data-set as been removed (in addition to the air).


(a)

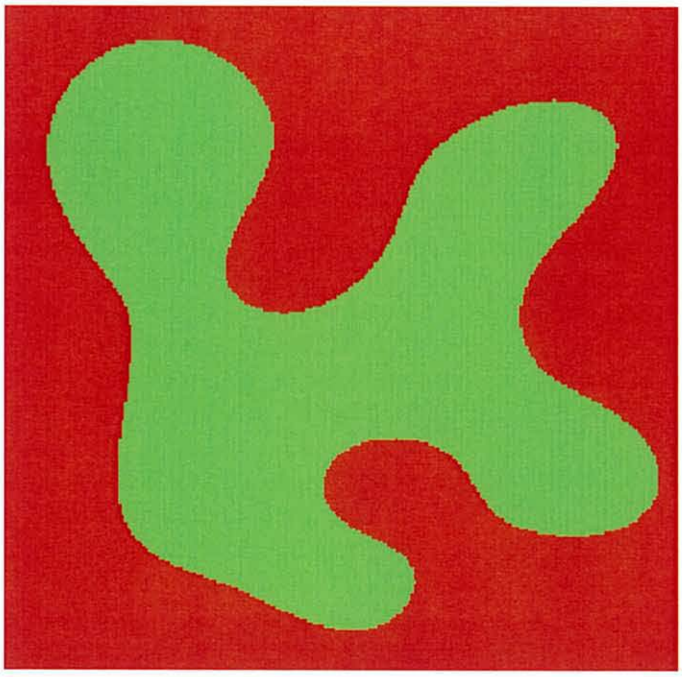
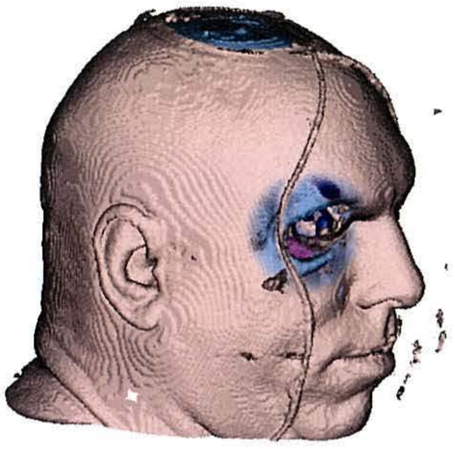
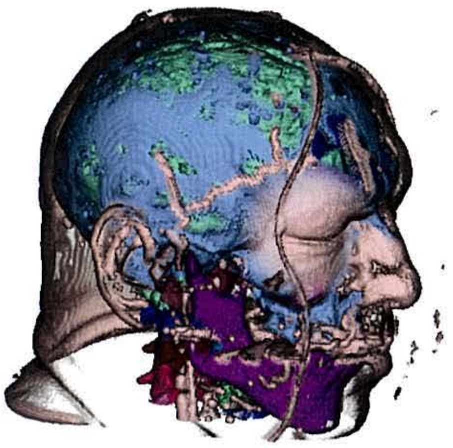

(b)

Figure 5.11: Classification results for test case. Prediction time for 256^3 voxels: 2.12 seconds. Number of vectors 491 (margin), 30 (error), 1377 (reserve). Feature space dimensions: 13


(a)


(b)

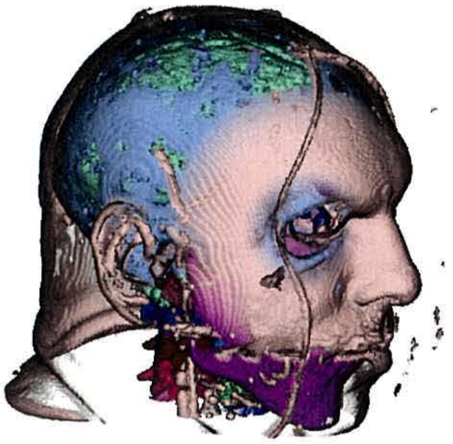

(c)

Figure 5.12: Focus region opacity modulation with context-preserving rendering. The focus point is the left eye. (a) samples become more opaque further away from the focus point in (a) while they become more transparent in (b). (c) is a combination of both (a) and (b) were the opacity increase the further away from the focus point, but then decrease outside the focus region.

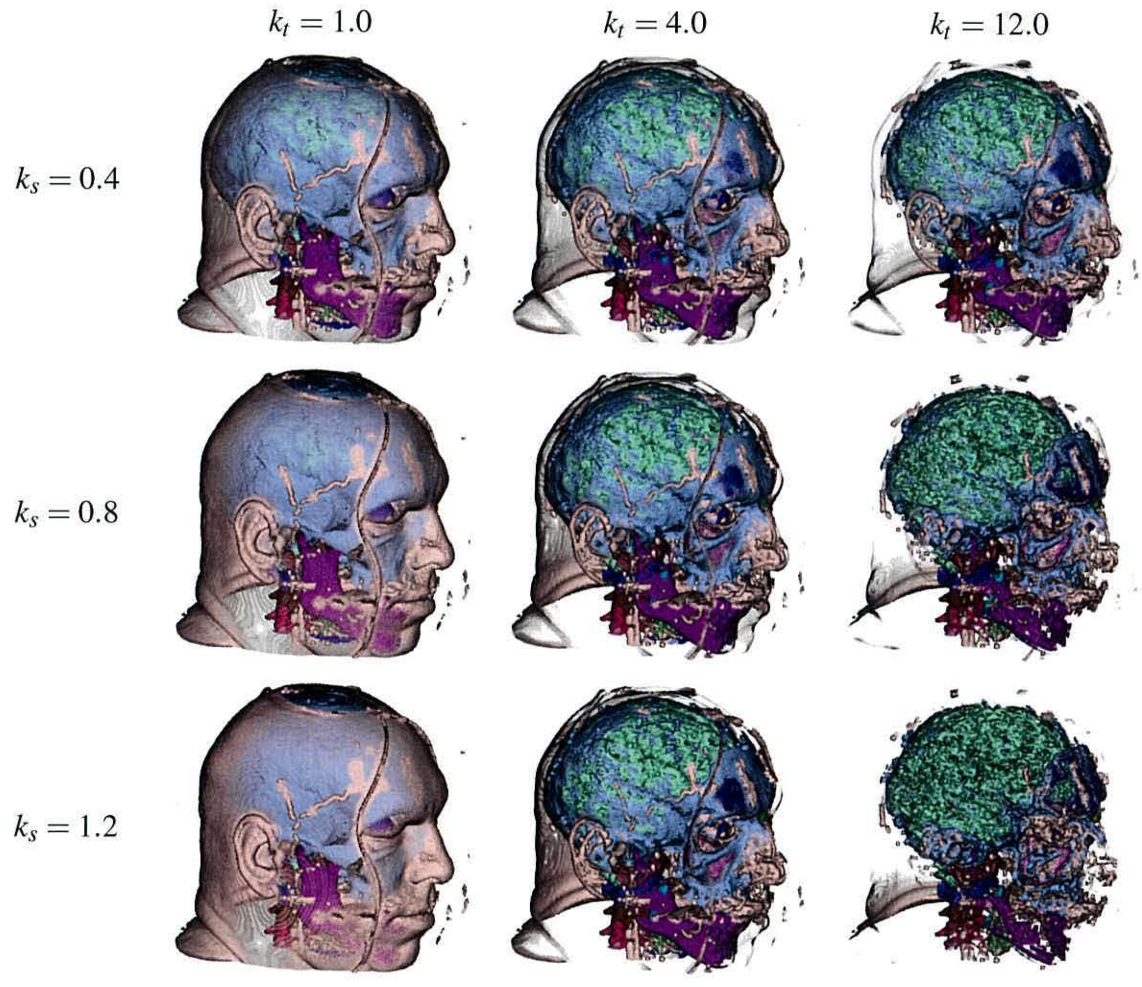
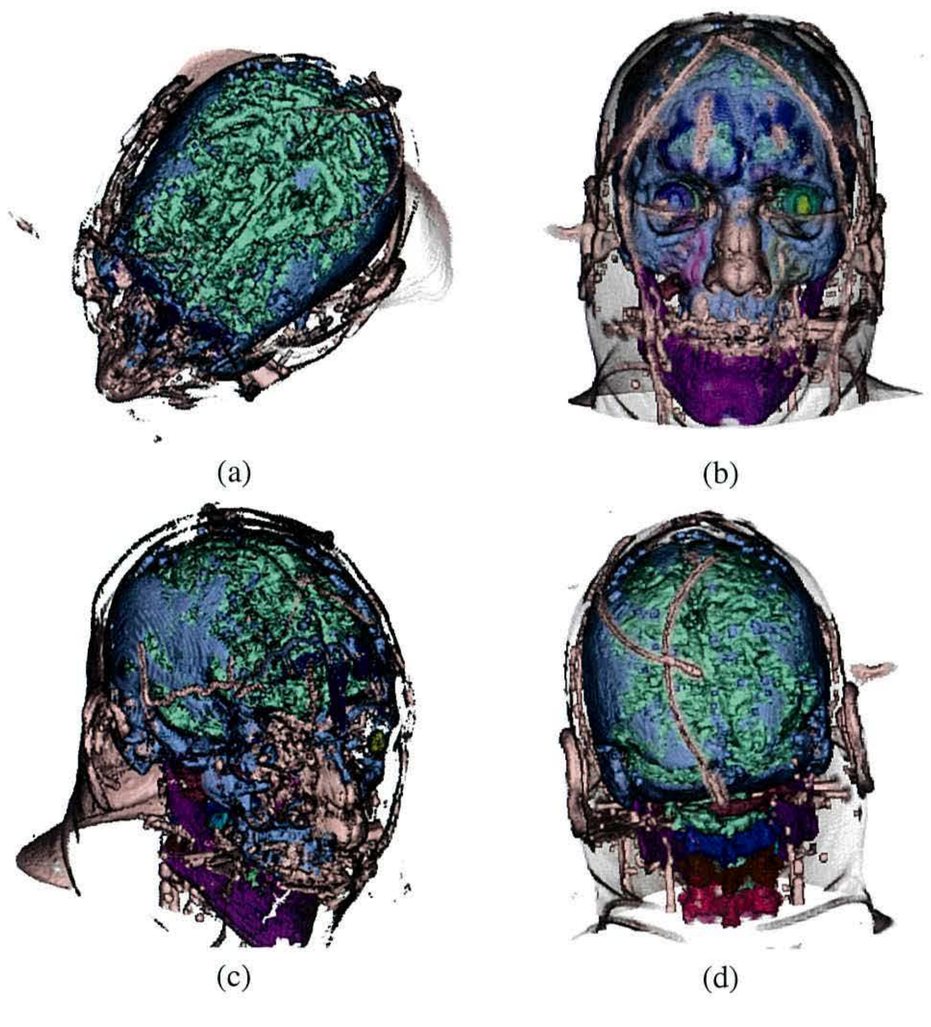


Figure 5.13: Context-preserving volume rendering of volumetric segmentation data using different values for k_t and k_s



(a) (b) (c) (d)

Figure 5.14: Renderings from different view points showing the context-preserving rendering of segmentation data

Chapter 6

Consistent Reconstruction of Surfaces

6.1 Introduction

Consistent results are a fundamental aspect of computer science. Indeed without a robust method, research may not be entirely reproducible in a consistent and reliable manner. For many years octrees have been employed for a magnitude of tasks, ranging from space partitioning to data down-sampling. For such an important tool it is vital that its use is consistent and reliable. However, as we show in this chapter, it apparent that even recent research can overlook inconsistencies due to rotation variance resulting from the use of an octree.

By design the octree is used to partition space into octants, which can recursively be subdivided. Typically, when applied to a space with some form of spatial data, the octree is centered and bound to that data; this ensures efficiency, and is widely practiced. While the sampling of an octree that has been centered and bound to the data is both position- and scale-invariant, it is not rotation-invariant. This is to say that if the data was scaled or translated, the octree results would be identical but not if the data was rotated.

In this thesis we explore the important field of surface reconstruction and highlight recent research where the rotation-variance of octrees has not been addressed. We provide a method utilizing PCA (Principle Component Analysis) by which rotation-invariance can be achieved for an octree employed for surface reconstruction. In addition we show the inconsistency of the previous method and the consistency of our approach using curvature analysis. Finally, a discussion is given as to how this method could be applied to volumetric data.

6.1.1 Previous Work

Octrees are have been used extensively in surface reconstruction, from simply pre-processing input data as done by Kalaiah [148], to allowing efficient handling of large data-set for surface reconstruction as reported by Kindlmann [156]. While this thesis only explores and proves one example of inconsistency resulting from the use of an octree, it is possible -but in no way asserted or proven here- that other methods employing an octree may be rotation-variant and subsequently inconsistent. The method explored in this thesis (as the case study) was reported by Ohtake [228], who used an an octree to build a hierarchy of points-sets for use in multi-scale RBF surface reconstruction.

Other examples of Octrees being used for surface reconstruction include the work by Dalmasso [76], who describes a volumetric approach to surface reconstruction from nonuniform data. The data volume is split and classified at different scales of spatial resolution into surface, internal and external voxels and hierarchically described using an octree.

An automatic and interactive system to repair both the shape and appearance of defective point sets by utilizing an octree was presented by Park [239]. Octree-based subdivision was employed by Ohtake [227] for large point sets to reconstruct surface models using multi-level partition of unity implicit surfaces, while Tobor [296] reconstructed multi-scale implicit surfaces with attributes, given discrete attributed point sets. Xie [331] organized sample points using an octree for a surface reconstruction and was able to recover high-quality surfaces from noisy and defective data sets without any normal or orientation information. Finally, Hornung [134] used a octree in a volumetric method for reconstructing watertight triangle meshes from arbitrary, unoriented point clouds.

6.2 Surface reconstruction

Surface reconstruction from unorganized point clouds is an important problem, especially for the recreation of real world objects that have been digitally scanned. Most object scanning technologies do not, by design, provide a surface model to be used instantly, but rather supply data by which a surface or an object can be recreated. There are a variety of sources from which data is obtained. Contour slices, where an object has been scanned using a CT scanner and an iso-surface has been defined, it a typical source. Another source is interactive tools, where data is created in real-time by a user. However, most prominent is range-data, where an object has been scanned using a laser to measure distances to the areas of an object.

There have been numerous solutions to the surface reconstruction problem. For examplem Hoppe [132] used an implicit surface model were surface reconstruction was defined as the zero set of an estimated signed distance function, while Bernardini [24] used a the rolling ball technique and Curless [75] used a volumetric approach.

Radial Basis Functions were used by Carr [51] to solve a scattered data interpolation problem and reconstruct surfaces. Alternatively, Nina Amenta [8, 7] used Voronoi vertices and Delaunay triangulation to create a piece-wise linear approximation of a smooth surface with better noise reduction.

Typically the problem of reconstructing a surface requires that the input data be converted to an unorganized point cloud in three-dimensional space. In this paper we are exploring an approach whereby the surface reconstruction task is cast as a scattered data interpolation problem and the reconstruction is defined as an implicit surface.

A surface that is not explicitly defined, but rather is embedded within another property, is called an implicit surface. A distance field is an example of an implicit surface. The surface of a distance field is typically defined as zero and all space exterior or interior to the surface is non-zero. Surface reconstructions based on implicit surfaces is a popular approach due to a number of advantages it has over other representations. A particularly noteworthy advantage is the ability of implicit surfaces to easily represent models of complex topology.

A well known approach for solving the scattered data interpolation problem is a RBF (Radial Basis Function) Network.

6.2.1 Radial Basis Functions

For the scattered data point interpolation, a RBF network is defined as [250, 125]

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i \phi\left(||\mathbf{x} - \mathbf{x}_i||\right) \tag{6.1}$$

which satisfies the interpolation conditions $f(\mathbf{x}_i) = y_i$ where $\mathbf{x}_i \in \mathbb{R}^3$ are data points, and $y_i \in \mathbb{R}$ are function values. Unlike height-function interpolation, a surface embedded in three-dimensional space is often defined as a zero-level set $f(\mathbf{x}) = 0$.

To avoid the trivial solution that f is zero everywhere, off-surface points are typically appended to the input data and are given non-zero values $y_i \neq 0$ whilst the on-surface points are defined with $y_i = 0$ [82, 299]. The coefficients w_i are determined by solving a linear system $\mathbf{Gw} = \mathbf{y}$ which is obtained by inserting the interpolation conditions into Eq. 6.1.

If the matrix $\mathbf{G}$ is full, however, this approach is limited to a small data set; approximately a thousand points or so. Given a large data set, a naive approach is to use a small subset of it and discard the remaining data points [82]. A better approach is to use CSRBFs (Compactly Supported RBFs) since their compact supports lead to a sparse linear system suitable for a large data set [33]. However, it is sensitive to the density of scattered data and, therefore, a careful selection of the support size for CSRBFs is required in surface reconstruction [33].

6.2.2 Multi-Layer Radial Basis Functions

To get around the CSRBFs problems, multi-level interpolation with a point hierarchy was proposed by Ohtake, *et al* [228]. Given a set of points $\mathcal{P} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ sampled from a smooth surface, the multi-scale hierarchy of point sets $\{\mathcal{P}^1, \mathcal{P}^2, \ldots, \mathcal{P}^M = \mathcal{P}\}$ is first constructed by spatial down sampling. Then the multi-level interpolation procedure proceeds in a coarse-to-fine way with decreasing support sizes. It recursively determines the set of interpolating functions $f^k(\mathbf{x}) = f^{k-1}(\mathbf{x}) + o^k(\mathbf{x})$ such that $f^k(\mathbf{x}) = 0$ interpolates $\mathcal{P}^k$ for $k = 1, 2, \ldots, M$ and $f^0(\mathbf{x}) = -1$. The offsetting function $o^k(\mathbf{x})$ has the form of

$$\sum_{\mathbf{p}_i \in \mathcal{P}^k} [g_i(\mathbf{x}) + w_i]\, \phi_\sigma(\|\mathbf{x} - \mathbf{p}_i\|) \tag{6.2}$$

where $g_i(\mathbf{x})$ are local polynomial approximations determined via least square fitting to $\mathcal{P}^k$ and $\phi_\sigma(\|\mathbf{x}-\mathbf{p}_i\|)$ are CSRBFs. The coefficients w_i are found by solving a linear system

$$\Phi w = -(\mathbf{f}+\mathbf{g}) \tag{6.3}$$

obtained by the interpolation conditions $f^k(\mathbf{p}_j) = 0$ for every point $\mathbf{p}_j \in \mathcal{P}^k$. The point hierarchy is created using octree-based subdivision. It starts with an axis-aligned box that encompasses the point set $\mathcal{P}$, and is followed by recursive subdivision of the space and points into eight octants or cells. $\mathcal{P}$ is clustered with respect to the cells by computing centroids of the points in each cell.

Depending on the coordinate system used to represent the points, however, it can lead to inconsistent surface reconstruction and geometry. For example, surface curvatures are important for matching and registration tasks and can result in different values even with the same point set if represented in different coordinate systems. An actual research example, Hadwiger [117], not only uses an octree for hierarchical representation of a volume, but also explicitly extracts curvature for visualisation, and it is possible the variance problem affects it too.

The variance, attributed to the coordinate system, is due to the octree subdivision such that each side of the cells is parallel to an axis of the coordinate system: rotation is especially problematic, whilst the octree is invariant to other coordinate transforms such as translation, scaling and flipping.

6.3 Consistent Surface Reconstruction using PCA

Ohtake's [228] multi-layer approach does not address the rotation-variance inherent to octree down-sampling. Indeed we prove later that an arbitrarily aligned octree results in inconsistencies in the surface reconstruction. Such inconsistencies could result in difficulty during object mark-up and reconstruction comparison, where consistency is vital.

To solve the problem of rotation-variance we turn to PCA (Principle Component Analysis). PCA involves a linear transformation of a data-set, such that the first principle component is the data-set projection with the greatest variance, the second principle component is the second greatest variance and so on. The first three principle components can be viewed as the cartesian axis defining the intrinsic orientation of the data. The appeal of this method is that any coordinate transformations applied to the data set will also effect its intrinsic orientation.

We arrive at an orthogonal coordinate system (the intrinsic-orientation of the data) from calculating the eigenvectors of the covariance matrix

$$\mathbf{C} = \mathbf{D}\mathbf{D}^T$$

where $\mathbf{D} = [\mathbf{x}_1 - \overline{\mathbf{x}}, \ldots, \mathbf{x}_N - \overline{\mathbf{x}}]$ and $\overline{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$. We then orientate the octree to the data using this coordinate system.

This approach is different than that of Kalaiah [148], where PCA is used on a group of points structured in an octree and used to determine the *local* orientation frame of the group, as we are re-orientating the *entire* octree *prior* to space partitioning and centroid calculation. In the work by Kalaiah [148] rotation-variance of octrees was not addressed either.

PCA has been widely used for the representation of shape, appearance and motion in the computer graphics field. For example, Sloan [286] compressed the storage, and accelerated performance, of pre-computed radiance transfer (PRT), which captures the way an object shadows, scatters, and reflects light, using clustered principal component analysis. Compressed representation of lighting information was also achieved using PCA in the work by Kristensen [167]. Other uses of PCA include, human face recognition (Feng [93]), and silhouette recognition (Gouaillier [105]). In addition, Torre [297] provided *Robust PCA* (RPCA) for computer vision that improved PCA representation of shape, appearance, and motion.

6.4 Flexible Basis Functions

For a wider choice of basis functions, we can use an approximation scheme. In the aforementioned interpolation approach, the down-sampled points $\mathbf{p}_j \in \mathcal{P}^k$ serve as both the basis centres and the data points as in Eq. 6.2 and 6.3, and only a few types of functions can make the linear system in Eq. 6.3 solvable. In the proposed approximation approach, we use $\mathcal{P}^k$ only for the basis centres and $\mathcal{P} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ for the data points. In this approximation scheme, we may obtain a linear system equivalent to Eq. 6.3 from the conditions $f^k(\mathbf{x}_i) = 0$. However, it is over-determined since the number of the basis functions is less than that of the data points, *i.e.*, $|\mathcal{P}^k| < |\mathcal{P}|$. As the least square solution, instead, we can determine the weights w_i by solving the following linear system

$$\Phi^T \Phi \mathbf{w} = -\Phi^T (\mathbf{f} + \mathbf{g}) . \tag{6.4}$$

In choosing the basis functions, this approximation scheme provides more flexibility than the interpolation does since there are more functions making Eq. 6.4 solvable than those for Eq. 6.3.

6.5 Experiments

In order to show the consistency of our method we analyzed the curvature of the reconstructed models. We used the curvature calculation method provided by Kindlmann [156] to calculate the mean curvature at each center.

First we formed two copies a 54K point-set, sampled from squirrel a data-set, and applied a rotation of 70 degrees around the y-axis to one of them. We then applied the PCA-Octree reorientation step on both data sets. After fitting two multi-scale CSRBFs, we calculated the mean curvature for each data-set. Figure 6.1 shows the curvature residual of two experiments: one with the PCA-Octree orientation step and one without. When the octrees are aligned with the input coordinate systems as in [228], they show noticeable discrepancies in these curvature values: 8.2 on average. When aligned with the PCA coordinate system as proposed, they show virtually no discrepancies: $1.0 * 10^{-4}$, of which attributed to numerical inaccuracy.;

We also experiment with the approximation scheme for the reconstruction and compare its results with that of the interpolation (Fig. 6.2). In addition to the aforementioned CSRBFs, we use simpler basis functions $\phi(r) = \left(1 - \frac{r^2}{\sigma^2}\right)_+^4$ which would be not usable in the interpolation scheme. These basis functions only need to compute r^2, but not r which would involve expensive computation of $sqrt()$ as in the CSRBFs whilst the reconstruction qualities are comparable.

6.6 Summary

This chapter has reviewed the utilization of octrees and PCA in current research. An exploration of leading research shown that the rotation-variance inherent to octrees when partition a space and spatial data was not being accounted for. By examining Ohtake's [228] multi-scale CSRBFs surface reconstruction technique a method for rotation-invariance by utilizing PCA was presented. It was shown that employing the PCA-Octree method produces consistent reconstructions. To evaluate and prove the claim, curvature analysis was performed and results compared. It was shown that the presented method resulted in consistency in surface reconstruction of arbitrarily-orientated data. Finally, this chapter also introduced flexibility to multi-scale CSRBFs by employing RBF approximation.

The approach described in this chapter could also be applied to volumetric data. For example, multiple CT or MRI scans of a patient may have minor variances in regard to rotation. Subsequently, space-division using an Octree or Kd-tree and the algorithms which depend on these hierarchy structures may have rotation variance in results. The application of PCA to acquire the data-variance and the transformation of the volume data such that the 3-principle components are aligned to the axis of the regular grid would help ensure more consistent results when comparing patient data. A typical scenario might be a patient with a brain tumor and multiple CT scans over many months, where minor changes in head position might result in rotation variance. Alternatively,

As CT data is typically reconstructed using the Radon Transform [48], future research might be to analyze the scanned data prior to the application of the inverse Radon transform and the discrete storage of the resulting volume in a regular grid.

This would help to reduce numerical error, which might be a problem if an existing volume was simply re-sampled along new axes; i.e. in order to align the data.

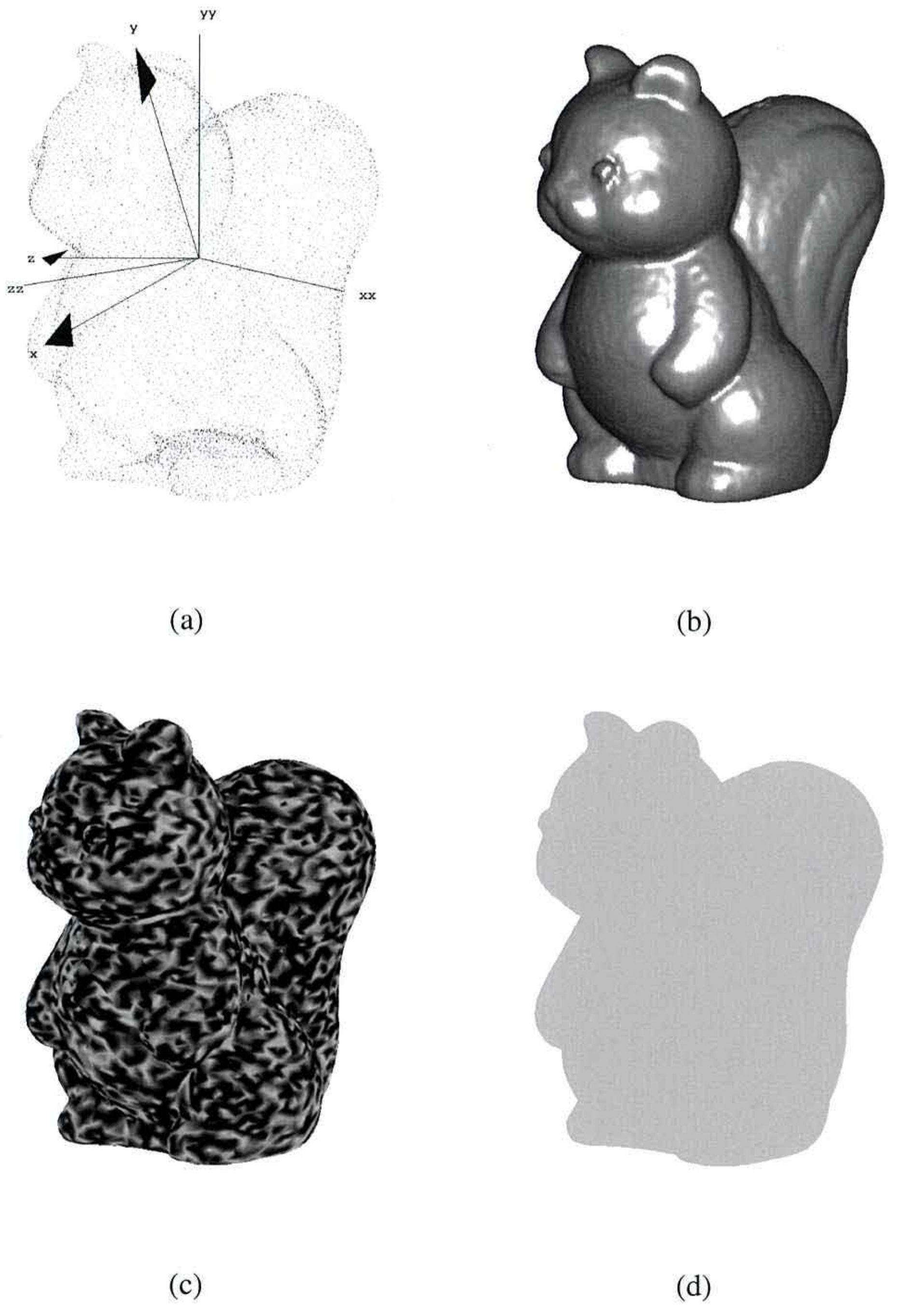
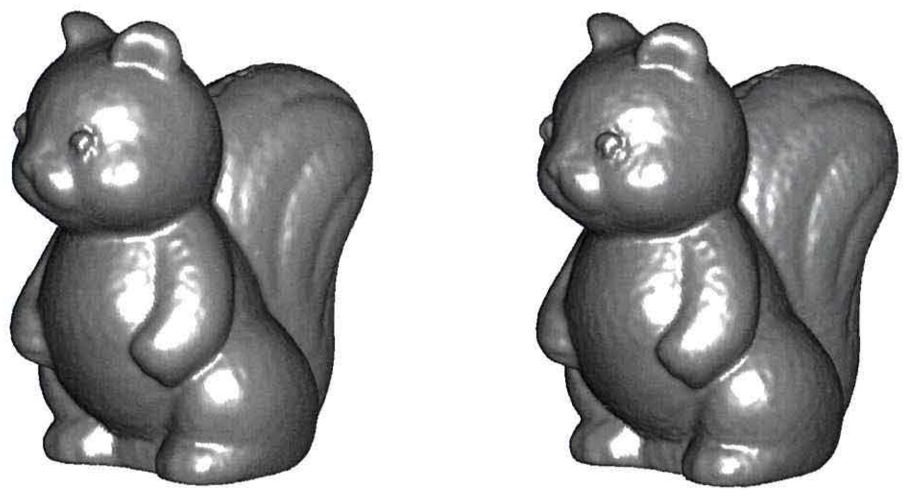


Figure 6.1: Surface reconstruction invariant to coordinate transforms. (a) the input data set represented by two different coordinate systems; (b) an example of surface reconstruction; (c) discrepancies in mean curvatures between the reconstructed surfaces with the octrees aligned to the input coordinate systems (the darker, the wider discrepancy); (d) discrepancies when aligned to the PCA coordinate system.



(a) (b)

Figure 6.2: Interpolation *vs.* approximation. Surface reconstructions using (a) the interpolation and (b) the approximation scheme both using $\phi(r) = \left(1 - \frac{r}{\sigma}\right)_+^4 \left(4\frac{r}{\sigma} + 1\right)$

Chapter 7

Conclusions

This chapter summarizes the thesis and outlines the original contributions.

7.1 Kd-Jump

This thesis has presented Kd-Jump, a stackless traversal of implicit kd-trees for faster isosurface raytracing. It was shown that Kd-Jump can outperform both stackless and stack-based approaches, while only needing a fraction of memory compared to a stack-based approach. Further, Kd-Jump exploits the index-based referencing used for implicit kd-trees to achieve traversal-paths equivalent to a stack-based method, without incurring the extra node visitation of kd-restart.

To further strengthen kd-tree, this work introduced Hybrid Kd-Jump. Hybrid Kd-Jump utilises a volume stepper for leaf testing and a run-time depth threshold to define where kd-tree traversal stops and volume stepping occurs. By using both methods one gains the benefits of empty-space removal and hardware-based texture interpolation. It was shown that Hybrid Kd-Jump performs well at removing empty space and can outperform a brute-force ray-tracer.

Memory usage for an implicit kd-tree may be too large if min/max pairs are stored in each node. This work has shown that, if the conditions for the current isosurface are moved out of traversal and into the tree nodes themselves, then significantly less memory is required. In addition, even with a naive implementation, updating the implicit kd-tree for a large volume was shown to be quite fast on modern GPU architecture.

Further, the usefulness of loading new rays once a warp of threads completes was reported and shows promising results for faster ray tracing. Additionally, this work discussed and examined the separation of the ray-tracing pipeline into separate kernels, and showed that the methodology has some promise for better efficiency.

7.2 Real-time Segmentation

A tool for real-time semi-automatic segmentation was presented. The tool utilized a support vector machine, which was trained with user generated input. The input was captured from a paint plane, whereby the user would paint upon a 2D visualization of a volume slice, either classifying voxels as wanted or unwanted features. Painted voxels where then converted to input vectors and given to the SVM for training. Once the SVM was trained, the remaining volume was segmented into wanted or unwanted by using the SVM to predict the volume voxel classes.

Real-time segmentation was attained by using incremental SVM. Incremental SVM allows new inputs to be trained on the fly, without the need to retrain the entire SVM with all inputs. This allows a user to progressively define segmentation objects over time, without there being a noticeable delay once painting stops. In addition, the work included an input labeling mechanism, which allowed for easier management of input vectors, such that if a voxel input needed to be unlearned then corresponding support vector in the SVM could be found.

Further, in order to ensure the class prediction of the volume did not cause a large delay, this work utilized the power of GPUs (CUDA). By using CUDA kernels, the high-parallel task of performing an SVM evaluation for each voxel was accelerated tenfold, when compared to a CPU implementation. There were two aspects accelerated by CUDA in regard to voxel class prediction: generation of the input vectors (i.e., converting a volume voxel into a test vector for use in the SVM) and the evaluation of the voxel vector with the SVM support vectors.

7.2.1 Context-Preserving Rendering

A visualization method for context-preservation was introduced for segmentation data. By directly visualising the segmentation data, rather than the underlying volume, it was shown that anatomical features could be observed. By only requiring the segmentation data, memory could be saved as only the boundaries of segmentation objects needed to be stored.

The work focused on the curvature information present at the boundaries and used this information for the context-preserving visualisation. By using the curvature information, derived from applying PCA to the boundary, it was shown that a user could adjust simple parameters in order to see deeper into the volume. For example, it was possible to see through the skin and skull of a human head in order to observe the brain. However, any features on the skull or skin with high context were still visible.

7.3 Consistent Surface Reconstruction

This work has reviewed the use of octrees and PCA in current research. This thesis explored, using a surface reconstruction case-study, how leading research does not account for the rotation-variance inherent to octrees when partition a space and spatial data. It was shown that not accounting for the problem produces inconsistent results within derived techniques. As a case-study, the surface reconstruction approach utilizing an octree and CSRBFs, was explored. .

The work examined Ohtake's [228] multi-scale CSRBFs surface reconstruction technique and presented a method for rotation-invariance by utilizing PCA. PCA was used to re-orientate the scattered data points prior to fitting; in-effect orientating the octree to the intrinsic-orientation of the data-set defined by the data variance. It was shown that employing the PCA-Octree method produces consistent reconstructions and consistent analysis results, such as curvature analysis, of arbitrarily-orientated data. In addition the work also introduced flexibility to multi-scale CSRBFs by employing RBF approximation. This work utilized RBF approximation and provided an example of a computationally inexpensive (relatively) compact support radial basis function, which was shown to produce similar results as the standard method.

7.4 Future Work

This section details two possible methods for future work, based on the work presented in this thesis. Also examined is the future direction other researchers have indicted in recent work.

7.4.1 View-Dependent Isosurface Rendering

One interesting outcome of the work detailed in this thesis, in relation to the Hybrid Kd-Jump method presented, is that GPU hardware is so fast that actually performing brute-force ray-tracing can be faster than using an acceleration structure; in certain situations. Future work based on this thesis will examine whether view dependence can lead to building an acceleration structure per-frame that is highly optimized for the topology of the local volume regions. Specifically, nodes where the isosurface runs tangential to the view rays and where the majority of rays will pass through empty space, can be further sub-divided. On the other hand, if the majority of rays are likely to intersect the isosurface, then we do not require sub-dividing the node. The main difficulty of this method is performing the BVH tree build in real-time per-frame and developing an heuristic to determine (or estimate) whether node-splitting should be performed.

7.4.2 Instantaneous Volumetric Feedback during SVM Training

The segmentation method detailed in this thesis allows a user to train an SVM using paint tools, such that the SVM is actually incrementally learning during the paint process. This facilitated far less delays when compared to a batch SVM learner. The main draw-back is that the volume cannot be visualised during the paint process, as performing the class prediction for each voxel is computationally expensive, even if accelerated by a GPU.

Future work will exploit the nature of SVMs in order to reduce the number of voxels needing SVM predictions. Specifically, by realizing that an SVM builds a hyperplane with a margin separating two regions, we can deduce that only the margin region will contain changes to the voxel class. As such a GPU-based margin/voxel follower could be developed. The method would only perform SVM predictions in the likely places that will define the boundary, in the remaining areas, a simple flood fill will suffice. Seed points for the algorithm would be the margin-vector set provided by the Incremental SVM model.

With such a method it would be possible to visualize the training process, as the user paints training data, using volume rendering. An alternative method would be to only concern ourselves with the absolute boundary and not flood-fill the remaining areas. Instead, we can apply the method for context-preserving rendering, as presented in this thesis, to visualize the boundary information, as only areas with curvature and gradient changes contribute to the visualization; i.e. areas inside the objects, as defined by the segmentation ID, have no gradient.

7.4.3 Overview of Research Trends

It is likely that DVR will see further enhancements for speed, with the visual information portrayed and accuracy when combining other methods, such as Isosurface visualization [162] and Ambient Occlusion [214]. Research by Knoll [162] already provides a method to accurately detect sharp peaks in the transfer function such that thin-surfaces are correctly rendered in DVR. Further research may be to incorporate affine arithmetic, which is computational expensive, to improve intersection results and further improve accuracy.

Advances in the rendering of implicit surfaces is also likely, as reported in the leading research by Singh [285]. In their work they report several research directions, ranging from incorporating image-space acceleration techniques and exploiting ray coherence using ray beams. Also discussed is the open problem of Ray-tracing parametric surfaces, such as the 18th degree polynomial resulting from applying Kajiya's [145] technique to bicubic surfaces.

The ray tracing of point-based models has also seen renewed interest. A recent paper by Kashyap [150] details an improved method for large data; although at only interactive frame rates. Improving speed appears to be an ongoing challenge in this area of visualization. The work shows the continued trend to increase data-size and visual complexity as computational power also increases.

Real-time construction of acceleration structures, rather than constructing them as a pre-process, appears to be a current tread for ray tracing. Specifically, animated scenes, where the geometry is constantly changing are likely to be further explored as a research problem. Recent advancements by Lauterbach [183] have shown that GPUs can be used for this task and that a robust system for building a scene graph and ray tracing it in real-time is achievable.

It is clear that ray tracing will appear to be useful in more applications, including games, in the next decade. NVidia have recently released a ray tracing API for their CUDA-enabled GPUs called Optix [224]. Like OpenRT [309], it provides a simple API for application developers to integrate ray tracing into their software for high-quality visualization and effects. Future developments are likely to be either integration of ray tracing methods into existing graphics APIs, such as DirectX or OpenGL, or the formation of a new and standardized graphics API.

References

[1] The tao framework. `http://sourceforge.net/projects/taoframework/`, 2008. 112

[2] R. Adams and L. Bischof. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 641–647, 1994. 45

[3] M. Ahmed and A. Farag. Two-stage neural network for volume segmentation of medical images. *Pattern Recognition Letters*, 18(11-13):1143–1151, 1997. 47

[4] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proc. 1st ACM conference on High Performance Graphics*, pages 145–149. ACM, 2009. 27

[5] J. Amanatides. Ray tracing with cones. In *ACM SIGGRAPH Computer Graphics*, pages 129–135. ACM, 1984. 24

[6] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987. 26, 36

[7] N. Amenta and M. Bern. Surface reconstruction by voronoi filtering. *Discrete and Computational Geometry*, 22:481–504, 1999. 131

[8] N. Amenta, M. Bern, and M. Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *ACM SIGGRAPH Computer Graphics*, pages 415–421, 1998. 131

[9] M. Amin, A. Grama, and V. Singh. Fast volume rendering using an efficient, scalable parallel formulation of the shear-warp algorithm. In *Proc. IEEE Symposium on Parallel Rendering*, pages 7–14. ACM, 1995. 34

[10] K. Anagnostou, T. Atherton, and A. Waterfall. 4D volume rendering with the Shear Warp factorisation. In *IEEE Transactions on Visualization and Computer Graphics*, pages 129 – 137. ACM, 2000. 35

[11] S. Anderson. Bit twiddling hacks. `http://graphics.stanford.edu/~seander/bithacks.html`, 2005. 77, 92

[12] N. Andrysco and X. X Tricoche. Matrix Trees. In To Appear: *Eurographics/IEEE-VGTC Symposium on Visualization*, volume 29. 54

[13] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968. 8, 24

[14] S. Armato, F. Li, M. Giger, H. MacMahon, S. Sone, and K. Doi. Lung Cancer: Performance of Automated Lung Nodule Detection Applied to Cancers Missed in a CT Screening Program. *Radiology*, 225(3):685–692, 2002. 44

[15] S. Armato III, M. Giger, and H. MacMahon. Automated detection of lung nodules in CT scans: preliminary results. *Medical Physics*, 28(8):1552–1561, 2001. 44

[16] S. Armato III and W. Sensakovic. Automated lung segmentation for thoracic CT:: Impact on computer-aided diagnosis. *Academic radiology*, 11(9):1011–1021, 2004. 44

[17] J. Arvo and D. Kirk. A Survey of Ray Tracing Acceleration Techniques. pages 201–262, 1989. 35

[18] P. Atherton, K. Weiler, and D. Greenberg. Polygon shadow generation. *ACM SIGGRAPH Computer Graphics*, 12(3):275–281, 1978. 24

[19] M. Atkins and B. Mackiewich. Fully automatic segmentation of the brain in MRI. *IEEE Transactions on Medical Imaging*, 17(1):98–107, 1998. 44

[20] C. Bajaj, V. Pascucci, D. Thompson, and X. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proc. IEEE symposium on Parallel visualization and graphics*, pages 97–104. IEEE Computer Society, 1999. 12

[21] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. pages 94–103, 2010. 40

[22] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. 36

[23] J. Bentley and J. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979. 38

[24] F. Bernardini, J. Mittleman, H. Rushmeir, and C. Silva. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Vision and Computer Graphics*, 5(4), 1999. 130

[25] J. Bezdek, R. Ehrlich, et al. FCM: The fuzzy c-means clustering algorithm. *Computers & Geosciences*, 10(2-3):191–203, 1984. 47

[26] B. Bilgiç. *Fast Human Detection with Cascaded Ensembles*. PhD thesis, Massachusetts Institute of Technology, 2010. 40

[27] A. Bleau and L. Leon. Watershed-based segmentation and region merging. *Computer Vision and Image Understanding*, 77(3):317–370, 2000. 46

[28] J. Blinn. Models of light reflection for computer synthesized pictures. *ACM SIGGRAPH Computer Graphics*, 11(2):192 – 198, 1977. 56

[29] J. Blinn. Simulation of wrinkled surfaces. In *ACM SIGGRAPH Computer Graphics*, pages 286–292. ACM, 1978. 24

[30] J. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *ACM SIGGRAPH Computer Graphics*, pages 21–29. ACM, 1982. 24, 25

[31] J. Blinn and M. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976. 24

[32] D. Blythe. The direct3d 10 system. In *ACM SIGGRAPH Computer Graphics*, page 734. ACM, 2006. 39

[33] B.Morse, T.Yoo, and D.Chen. Interpolating implicit surfaces from scattered surface data using compactly supported radial basis functions. In *Shape Modeling & Applications*, pages 89–98, 2001. 132

[34] B. Boser, I. Guyon, and V. Vapnik. A training algorithm for optimal margin classifiers. In *Workshop on Computational Learning Theory*, pages 144–152. ACM, 1992. 99

[35] J. Bouknight and K. Kelley. An algorithm for producing half-tone computer graphics presentations with shadows and movable light sources. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, pages 1–10. ACM, 1970. 24

[36] H. Bourquain, A. Schenk, F. Link, B. Preim, G. Prause, and H. Peitgen. HepaVision2: A software assistant for preoperative planning in living-related liver transplantation and oncologic liver surgery. *Computer Assisted Radiology and Surgery*, pages 341–346, 2002. 43

[37] S. Bruckner, S. Grimm, and A. Kanitsar. Illustrative Context-Preserving Exploration of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1559–1569, 2006. 23, 42, 43, 117

[38] S. Bruckner, S. Grimm, A. Kanitsar, and M. Gröller. Illustrative context-preserving volume rendering. In *Proc. of EuroVis 05*, pages 69–76, 2005. 54

[39] S. Bruckner and M. Groller. Volumeshop: An interactive system for direct volume illustration. In *IEEE Transactions on Visualization and Computer Graphics*, volume 5, pages 671–678, 2005. 56

[40] S. Bruckner and M. Groller. Style transfer functions for illustrative volume rendering. In *Computer Graphics Forum*, volume 26, pages 715–724. Blackwell Publishing, 2007. 42

[41] M. Brummer, R. Mersereau, R. Eisner, and R. Lewine. Automatic Detection of Brain Contours in MRI Data Sets. In *Proceedings of the 12th International Conference on Information Processing in Medical Imaging*, page 204. Springer-Verlag, 1991. 44

[42] D. Bucciarelli. Smallptcpu vs smallptgpu. `http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html`, 2010. 40

[43] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH Computer Graphics*, page 786. ACM, 2004. 40

[44] M. Bunnell. Dynamic ambient occlusion and indirect lighting. *GPU Gems*, 2:223–233, 2005. 24

[45] C. Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998. 99

[46] H. Byun and S. Lee. Applications of support vector machines for pattern recognition: A survey. *Pattern Recognition with Support Vector Machines*, 2002. 59

[47] J. Caban and P. Rheingans. Texture-based Transfer Functions for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1364–1371, 2008. 60

[48] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proc. Volume Visualization*, pages 91–98. ACM, 1994. 31, 137

[49] P. Campadelli and E. Casiraghi. Liver Segmentation from CT Scans: A Survey. In *Proc. 7th international workshop on Fuzzy Logic and Applications*, pages 520–528. Springer-Verlag, 2007. 45

[50] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry: Theory and Applications*, 24(2):75–94, 2003. 44

[51] J. C. Carr and R. K. Beatson. Reconstruction and representation of 3d objects with radial basis functions. In *ACM SIGGRAPH Computer Graphics*, pages 67–76, 2001. 131

[52] N. Carr, J. Hall, and J. Hart. The ray engine. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Graphics hardware*, pages 37–46. Eurographics Association, 2002. 27

[53] N. Carr, J. Hoberock, K. Crane, and J. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209. Canadian Information Processing Society, 2006. 53

[54] J. Cates, A. Lefohn, and R. Whitaker. GIST: an interactive, GPU-based level set segmentation tool for 3D medical images. *Medical Image Analysis*, 8(3):217–231, 2004. 59

[55] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. *Advances in neural information processing systems*, pages 409–415, 2001. 102, 103

[56] S. Chan and E. Purisima. A new tetrahedral tesselation scheme for isosurface generation. *Computers & Graphics*, 22(1):83–90, 1998. 9

[57] C. Chang and C. Lin. LIBSVM: a library for support vector machines. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`, 2001. 106

[58] J. Charles, L. Kuncheva, B. Wells, and I. Lim. Stability of Kerogen Classification with Regard to Image Segmentation. *Mathematical Geosciences*, 41(4):475–486, 2009. 44

[59] H. Chen, F. Samavati, and M. Sousa. GPU-based point radiation for interactive volume sculpting and segmentation. *The Visual Computer*, 24(7):689–698, 2008. 60

[60] S. Chen and R. Radke. Level Set Segmentation with Both Shape and Intensity Priors. In *Proc. International Conference on Computer Vision*, 2009. 46

[61] W. Chen, L. Ren, M. Zwicker, and H. Pfister. Hardware-accelerated adaptive EWA volume splatting. In *Proceedings of IEEE Visualization 2004*, Oct. 2004. 32, 33

[62] Y. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. *IEEE Transactions on Visualization and Computer Graphics*, pages 217–224, 2003. 12

[63] Y. Chiang, C. Silva, and W. Schroeder. Interactive out-of-core isosurface extraction. In *IEEE Transactions on Visualization and Computer Graphics*, pages 167–174. IEEE Computer Society Press, 1998. 12, 14

[64] 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. 11

[65] M. Clark, L. Hall, D. Goldgof, L. Clarke, R. Velthuizen, and M. Silbiger. MRI segmentation using fuzzy clustering techniques. *IEEE Engineering in Medicine and Biology Magazine*, 13(5):730–742, 1994. 47

[66] L. Clarke, R. Velthuizen, M. Camacho, J. Heine, M. Vaidyanathan, L. Hall, R. Thatcher, and M. Silbiger. MRI segmentation: methods and applications. *Magnetic Resonance Imaging*, 13(3):343–368, 1995. 43

[67] H. Cline, W. Lorensen, R. Kikinis, and F. Jolesz. Three-dimensional segmentation of MR images of the head using probability and connectivity. *Journal of Computer Assisted Tomography*, 14(6):1037, 1990. 44

[68] I. Cohen, L. Cohen, and N. Ayache. Using deformable surfaces to segment 3-D images and infer differential structures. In *Computer Vision*, pages 648–652. Springer. 47

[69] D. Collins, C. Holmes, T. Peters, and A. Evans. Automatic 3-D model-based neuroanatomical segmentation. *Human Brain Mapping*, 3(3):190–208, 2004. 44

[70] C. Correa and K. Ma. Size-based Transfer Functions: A New Volume Exploration Technique. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1380–1387, 2008. 61

[71] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995. 101

[72] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Transactions on Visualization and Computer Graphics*, pages 235–243. IEEE Computer Society Press, 1997. 36

[73] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. 23, 28, 51

[74] T. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, 1994. 31

[75] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *ACM SIGGRAPH Computer Graphics*, pages 303 – 312, 1996. 130

[76] P. Dalmasso and R. Nerino. Hierarchical 3d surface reconstruction based on radial basis functions. In *3D Data Processing, Visualization and Transmission*, pages 574– 579, September 2004. 129

[77] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum*, volume 27, pages 1225–1233. Blackwell Publishing, 2008. 38

[78] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proc. 1992 workshop on Volume visualization*, pages 91–98. ACM, 1992. 17

[79] C. Diehl and G. Cauwenberghs. Incremental svm learning. `http://www.cpdiehl.org/incrementalSVM.html`, 2003. 105

[80] C. Diehl and G. Cauwenberghs. SVM incremental learning, adaptation and optimization. In *Proceedings of the 2003 International Joint Conference on Neural Networks*, pages 2685–2690, 2003. 103, 104, 105

[81] J. Diepstraten, D. Weiskopf, and T. Ertl. Transparency in Interactive Technical Illustrations. *Computer Graphics Forum*, 21(3):317–325, 2002. 43

[82] H. Q. Dinh, G. Turk, and G. Slabaugh. Reconstructing surfaces by volumetric regularization using radial basis functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(10):1358–1371, October 2002. 132

[83] L. Doctor and J. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics and Applications*, 1(3):29–38, 1981. 37

[84] R. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *ACM SIGGRAPH Computer Graphics*, pages 65–74. ACM New York, NY, USA, 1988. 20, 25, 41

[85] R. Duda, P. Hart, and D. Stork. *Pattern classification*. Wiley Interscience, 2001. 47

[86] M. Durst. Letters: Additional reference to marching cubes. *Computer Graphics*, 22(2):72–73, 1988. 9

[87] D. Ebert and P. Rheingans. Volume illustration: Non-photorealistic rendering of volume models. *IEEE Visualization 2000 Proceedings*, pages 195–202, 2000. 8, 43

[88] T. Elvins. A survey of algorithms for volume visualization. *ACM SIGGRAPH Computer Graphics*, 26(3):194–201, 1992. 23

[89] K. Engel, M. Hadwiger, C. Rezk-Salama, and J. Kniss. *Real-time volume graphics*. AK Peters Ltd, 2006. 16, 19, 23

[90] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16. ACM New York, NY, USA, 2001. 11, 14, 18, 21, 54

[91] A. Falcao and F. Bergo. Interactive volume segmentation with differential image foresting transforms. *IEEE Transactions on Medical Imaging*, 23(9):1100–1108, 2004. 46

[92] E. Farrell. Color display and interactive interpretation of three-dimensional data. *IBM Journal of Research and Development*, 27(4):356–366, 1983. 20

[93] G. C. Feng, P. C. Yuen, and D. Q. Dai. Human face recognition using pca on wavelet subband. *Journal of Electronic Imaging*, 9(2):362, 2000. 135

[94] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22. ACM, 2005. 28, 30, 50, 51, 52, 57

[95] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU raytracer. In *ACM SIGGRAPH Computer Graphics*, pages 15–22. ACM, 2005. 37, 53

[96] H. Friedrich, I. Wald, J. Gunther, G. Marmitt, and P. Slusallek. Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, 2007. 13, 23

[97] K. Fu and J. Mui. A survey on image segmentation. *Pattern recognition*, 13(1):3–16, 1981. 43

[98] H. Fuchs, Z. Kedem, and S. Uselton. Optimal surface reconstruction from planar contours. *Communications of the ACM*, 20(10):693–702, 1977. 9

[99] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, pages 16–26, 1986. 26, 36

[100] A. Glassner. *Principles of digital image synthesis*. Morgan Kaufmann, 1995. 16

[101] E. Gobbetti, F. Marton, and J. Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008. 36

[102] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987. 37

[103] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *ACM SIGGRAPH Computer Graphics*, pages 447–452, 1998. 56

[104] C. Goral, K. Torrance, D. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics*, 18(3):213–222, 1984. 24

[105] V. Gouaillier, L. Gagnon, and G. T. Andrew. Ship silhouette recognition using principal components analysis. In *Applications of digital image processing*, volume 3164, pages 59–69, 1997. 135

[106] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. *Proc. IEEE/ACM Supercomputing*, pages 89–99, 2006. 91

[107] M. Grob, C. Lojewski, M. Bertram, and H. Hagen. Fast implicit kd-trees: Accelerated isosurface ray tracing and maximum intensity projection for large scalar fields. In *Computer Graphics and Imaging*, pages 67–74, 2007. 13, 55

[108] A. Gueziec and R. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):328 – 342, 1995. 9

[109] J. Gunther, S. Popov, H. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *IEEE Symposium on Interactive Ray Tracing*, pages 113–118, 2007. 38

[110] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118, Sept. 2007. 38

[111] L. Guo and X. Mei. Implementation and Improvement Based on Shear-Warp Volume Rendering Algorithm. In *Computer Engineering and Technology*, volume 1, 2009. 35

[112] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. *IEEE Transactions on Visualization and Computer Graphics*, pages 53–60, 2002. 37

[113] R. Haber and D. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. *Visualization in Scientific Computing*, pages 74–93, 1990. 8

[114] M. Hadwiger, C. Berger, and H. Hauser. High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware. IEEE Computer Society Washington, DC, USA, 2003. 22, 39, 61, 117

[115] M. Hadwiger, C. Langer, H. Scharsach, and K. Bhler. State of the Art Report 2004 on GPU-Based Segmentation. *Technical Report TR-VRVis-2004-017*, pages 409–415, 2004. 59, 97

[116] M. Hadwiger, F. Laura, C. Rezk-Salama, T. Hollt, G. Geier, and T. Pabel. Interactive Volume Exploration for Feature Detection and Quantification in Industrial CT Data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1507–1514, 2008. 54

[117] M. Hadwiger, C. Sigg, H. Scharsach, and K. Bhler. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Eurographics, year = 2005, pages = 303-312,*. 37, 133

[118] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005. 13, 14, 29, 62

[119] L. Hai-liang. Research on Digital Image Segmentation Techniques [J]. *Computer Knowledge and Technology*, 9, 2009. 59

[120] L. Hall, A. Bensaid, L. Clarke, R. Velthuizen, M. Silbiger, and J. Bezdek. A comparison of neural network and fuzzy clustering techniques insegmenting magnetic resonance images of the brain. *IEEE Transactions on Neural Networks*, 3(5):672–682, 1992. 47

[121] M. Hardisty, L. Gordon, P. Agarwal, T. Skrinskas, and C. Whyne. Quantitative characterization of metastatic disease in the spine. Part I. Semiautomated segmentation using atlas-based deformable registration and the level set method. *Medical physics*, 34:3127–3179, 2007. 47

[122] G. Harris, P. Barta, L. Peng, S. Lee, P. Brettschneider, A. Shah, J. Henderer, T. Schlaepfer, and G. Pearlson. MR volume segmentation of gray matter and white matter using manual thresholding: dependence on image brightness. *American Journal of Neuroradiology*, 15(2):225–230, 1994. 44

[123] V. Havran and J. Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, 2002. 37

[124] S. Haykin. *Neural networks: a comprehensive foundation.* Prentice Hall PTR Upper Saddle River, NJ, USA, 1994. 47

[125] S. Haykin. *Neural networks:a comprehensive foundation*, chapter 5, page 280284. Prentice Hall inc., 2nd. edition edition, 1999. 131

[126] X. He, K. Torrance, F. Sillion, and D. Greenberg. A comprehensive physical model for light reflection. In *ACM SIGGRAPH Computer Graphics*, pages 175–186. ACM, 1991. 7

[127] M. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf. SVMs - a practical consequence of learning theory. *IEEE Intelligent Systems and Applications*, 13(4):18–21, 1998. 99, 100

[128] P. Heckbert and P. Hanrahan. Beam tracing polygonal objects. In *ACM SIGGRAPH Computer Graphics*, page 127. ACM, 1984. 24

[129] K. Hoehne, R. Delapaz, R. Bernstein, and R. Taylor. Combined surface display and reformatting for the three-dimensional analysis of tomographic data. *Invest Radiol*, 22:658–664, 1987. 20

[130] K. Hohne, M. Bomans, A. Pommert, M. Riemer, C. Schiers, U. Tiede, and G. Wiebecke. 3D visualization of tomographic volume data using the generalized voxel model. *The Visual Computer*, 6(1):28–36, 1990. 26, 44

[131] S. Hojjatoleslami and J. Kittler. Region growing: A new approach. *IEEE Transactions on Image Processing*, 7(7):1079–1084, 1998. 45

[132] H. Hoppe, T.Derose, T. Duchamp, and J. Mcdonald. Surface reconstruction from unorganized points. In *ACM SIGGRAPH Computer Graphics*, pages 71 – 78, 1992. 130

[133] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive kd tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174. ACM New York, NY, USA, 2007. 28, 30, 49, 51, 52

[134] A. Hornung and L. Kobbelt. Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information. In *Eurographics Symposium on Geometry Processing*, pages 41–50, 2006). 130

[135] S. Hu, E. Hoffman, J. Reinhardt, et al. Automatic lung segmentation for accurate quantitation of volumetric X-ray CT images. *IEEE Transactions on Medical Imaging*, 20(6):490–498, 2001. 44

[136] A. Huang and G. Nielson. Thin structure segmentation and visualization in three-dimensional biomedical images: a shape-based approach. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):93–102, 2006. 62

[137] W. Hunt and W. Mark. Adaptive acceleration structures in perspective space. In *IEEE Symposium on Interactive Ray Tracing*, pages 11–17, 2008. 37

[138] C. Jackins and S. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980. 37

[139] S. Jaffey and K. Dutta. Digital perspective correction for cylindrical holographic stereograms. *Processing and display of three-dimensional data*, pages 130–140, 1983. 20

[140] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve. Fourier volume rendering on the GPU using a split-stream-FFT. In *Vision, Modeling and Visualization*, pages 395–403, 2004. 39

[141] H. Jensen. Global illumination using photon maps. *Rendering Techniques*, 96:21–30, 1996. 24

[142] M. Jones. The production of volume data from triangular meshes using voxelisation. In *Computer Graphics Forum*, volume 15, pages 311–318. John Wiley & Sons, 1996. 13

[143] T. Kadir and M. Brady. Unsupervised non-parametric region segmentation using level sets. In *Proc. Computer Vision*, pages 1267–1274, 2003. 44

[144] A. Kadosh, D. Cohen-Or, R. Yagel, G. CommerceZone, and I. Jerusalem. Tricubic interpolation of discrete surfaces for binary volumes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):580–586, 2003. 17, 62, 79

[145] J. Kajiya. Ray tracing parametric patches. *ACM SIGGRAPH Computer Graphics*, 16(3):254, 1982. 146

[146] J. Kajiya. The rendering equation. In *ACM SIGGRAPH Computer Graphics*, pages 143–150. ACM, 1986. 24

[147] J. Kajiya and B. Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH Computer Graphics*, 18(3):165–174, 1984. 25, 36

[148] A. Kalaiah and V. A. Statistical point geometry. In *Eurographics Symposium on Geometry Processing*, pages 107 – 115, 2003. 129, 134

[149] M. Kaplan. The use of spatial coherence in ray tracing. *Techniques for Computer Graphics*, pages 173–193, 1987. 27

[150] S. Kashyap, R. Goradia, P. Chaudhuri, and S. Chandran. Real time ray tracing of point-based models. In To Appear: *ACM SIGGRAPH Computer Graphics*. ACM, 2010. 146

[151] D. Kay and D. Greenberg. Transparency for computer synthesized images. *ACM SIGGRAPH Computer Graphics*, 13(2):158–164, 1979. 24

[152] T. Kay and J. Kajiya. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics*, 20(4):269–278, 1986. 35, 38

[153] D. Kennedy, P. Filipek, and V. Caviness. Anatomic segmentation and volumetric calculations in nuclear magnetic resonance imaging. *IEEE Transactions on Medical Imaging*, 8(1):1–7, 1989. 44

[154] E. Keppel. Approximating complex surfaces by triangulation of contour lines. *IBM Journal of Research and Development*, 19(1):2–11, 1975. 9

[155] G. Kindlmann and J. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proc. Volume Visualization*, pages 79–86. ACM, 1998. 41

[156] G. Kindlmann, R.Whitaker, T. Tasdizen, and T. Moller. Curvature-based transfer functions for direct volume rendering: methods and applications. *IEEE Transactions on Vision and Computer Graphics*, 5(4):513– 520, 2003. 129, 136

[157] J. Kloetzli, M. Olano, and P. Rheingans. Interactive volume isosurface rendering using BT volumes. In *Proc. Symposium on Interactive 3D graphics and games*, pages 45–52. ACM, 2008. 55

[158] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *IEEE Transactions on Visualization and Computer Graphics*, pages 255–262. IEEE Computer Society, 2001. 42

[159] J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, pages 270–285, 2002. 22, 42

[160] A. Knoll. A Survey of Octree Volume Rendering Methods. *Scientific Computing and Imaging Institute, University of Utah*, 2006. 37

[161] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *IEEE Symposium on Interactive Ray Tracing, 2007. RT'07*, pages 11–18, 2007. 54

[162] A. Knoll, Y. Hijazi, R. Westerteiger, M. Schott, C. Hansen, and H. Hagen. Volume Ray Casting with Peak Finding and Differential Sampling. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1571–1578, 2009. 14, 55, 146

[163] A. Knoll, I. Wald, and C. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 2009. 27, 36

[164] A. Knoll, I. Wald, and C. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 2009. 28

[165] A. Knoll, I. Wald, S. Parker, and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *IEEE Symposium on Interactive Ray Tracing 2006*, pages 115–124, 2006. 54

[166] A. Knoll, I. Wald, S.Parker, and C.Hansen. Interactive isosurface ray tracing of large octree volumes. In *IEEE Symposium on Interactive Ray Tracing (2006)*, pages 115–124, 2006. 37

[167] A. W. Kristensen, T. Akenine-Mller, and H. W. Jensen. Precomputed local radiance transfer for real-time lighting design. In *ACM SIGGRAPH Computer Graphics*, pages 1208 – 1215, 2005. 135

[168] A. Krueger, C. Kubisch, B. Preim, and G. Strauss. Sinus Endoscopy-Application of Advanced GPU Volume Rendering for Virtual Endoscopy. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1491–1498, 2008. 54

[169] J. Kruger, J. Schneider, and R. Westermann. ClearView: An interactive context preserving hotspot visualization technique. *IEEE Transactions on Visualization and Computer Graphics (Proc. Visualization/Information Visualization 2006)*, 12(5), 2006. 56, 119

[170] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003*, pages 287–292, 2003. 22

[171] J. Kruger and R. Westermann. Acceleration techniques for GPU-based volume rendering. *IEEE Visualization, 2003. VIS 2003*, pages 287–292, 2003. 28, 29, 39

[172] H. Kuhn and A. Tucker. Nonlinear programming. *ACM SIGMAP Bulletin*, pages 6–18, 1982. 103

[173] J. Kuhnigk, V. Dicken, L. Bornemann, A. Bakai, D. Wormanns, S. Krass, and H. Peitgen. Morphological segmentation and partial volume analysis for volumetry of solid pulmonary lesions in thoracic CT scans. *IEEE transactions on medical imaging*, 25(4):417–434, 2006. 44

[174] L. Kuncheva, J. Charles, N. Miles, A. Collins, B. Wells, and I. Lim. Automated kerogen classification in microscope images of dispersed kerogen preparation. *Mathematical Geosciences*, 40(6):639–652, 2008. 44

[175] P. Lacroute. Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. In *Proc. IEEE Symposium on Parallel Rendering*, pages 15–22. ACM, 1995. 34

[176] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *ACM SIGGRAPH Computer Graphics*, pages 451–458. ACM New York, NY, USA, 1994. 33, 34

[177] H. Ladak, J. Thomas, J. Mitchell, B. Rutt, and D. Steinman. A semi-automatic technique for measurement of arterial wall from black blood MRI. *Medical Physics*, 28(6):1098–1107, 2001. 45

[178] E. Lafortune. Mathematical models and monte carlo algorithms for physically based rendering. Technical report, Cornell University)., 1996. 24

[179] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proc. 1st international symposium on Non-photorealistic animation and rendering*, pages 13–20. ACM, 2000. 56

[180] E. LaMar, B. Hamann, and K. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Transactions on Visualization and Computer Graphics*, pages 355–361. IEEE Computer Society Press, 1999. 37

[181] T. Larsson and T. Akenine-Moller. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, 30(3):450–459, 2006. 39

[182] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM SIGGRAPH Computer Graphics*, 25(4):288, 1991. 33

[183] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. In *Computer Graphics Forum*, volume 28, pages 375–384. Blackwell Publishing, 2009. 39, 147

[184] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28:375–384, 2009. 92

[185] C. Ledergerber, G. Guennebaud, M. Meyer, M. Bacher, and H. Pfister. Volume MLS ray casting. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1372–1379, 2008. 17

[186] T. Lee, M. Cho, C. Shieh, P. Chao, and H. Chang. Precise Segmentation Rendering for Medical Images Based on Maximum Entropy Processing. *LECTURE NOTES IN COMPUTER SCIENCE*, 3683:366, 2005. 62

[187] A. Lefohn, J. Cates, and R. Whitaker. Interactive, gpu-based level sets for 3d segmentation. *Medical Image Computing and Computer-Assisted Intervention*, pages 564–572, 2003. 59

[188] V. Lempitsky, M. Verhoek, A. Noble, A. Blake, and A. Blake. Random Forest Classification for Automatic Delineation of Myocardium in Real-Time 3D Echocardiography. *echocardiography*, 2009. 59

[189] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8:29–37, 1988. 21, 25

[190] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics (TOG)*, 9(3):245–261, 1990. 21, 26, 35, 36

[191] M. Levoy and R. Whitaker. Gaze-directed volume rendering. In *Proc. Symposium on Interactive 3D graphics*, pages 217–223. ACM, 1990. 21

[192] W. Li, K. Mueller, and A. Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *IEEE Transactions on Visualization and Computer Graphics*, pages 42–50. IEEE Computer Society, 2003. 39

[193] B. Liu, G. Clapworthy, and F. Dong. Fast Isosurface Rendering on a GPU by Cell Rasterization. In *Computer Graphics Forum*, volume 28, pages 2151–2164. John Wiley & Sons, 2009. 55

[194] B. Liu, G. J. Clapworthy, and F. Dong. Multi-layer Depth Peeling by Single-Pass Hardware Rasterisation for Faster Isosurface Raytracing on a GPU. In To Appear: *Eurographics/IEEE-VGTC Symposium on Visualization*, volume 29. 55

[195] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *IEEE Transactions on Visualization and Computer Graphics*, pages 175–180. IEEE Computer Society Press, 1998. 11

[196] Y. Livnat, H. Shen, and C. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996. 10, 11

[197] W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM SIGGRAPH Computer Graphics*, pages 163–169. ACM New York, NY, USA, 1987. 9, 54

[198] M. Lorenzo-Valdes, G. Sanchez-Ortiz, R. Mohiaddin, and D. Rueckert. Segmentation of 4D cardiac MR images using a probabilistic atlas and the EM algorithm. *Medical Image Computing and Computer-Assisted Intervention*, pages 440–450, 2003. 47

[199] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. *IEEE Transactions on Visualization and Computer Graphics*, pages 211–218, 2002. 8

[200] D. Luebke and S. Parker. Interactive ray tracing with cuda. In *NVIDIA Technical Presentation, SIGGRAPH*. NVIDIA, 2008. 30, 40

[201] E. Lum, J. Shearer, and K. Ma. Interactive multi-scale exploration for volume classification. *The Visual Computer*, 22(9):622–630, 2006. 61

[202] J. MacDonald and K. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990. 30, 37

[203] J. Maintz and M. Viergever. A survey of medical image registration. *Medical image analysis*, 2(1):1–36, 1998. 43

[204] J. Maldjian, P. Laurienti, R. Kraft, and J. Burdette. An automated method for neuroanatomic and cytoarchitectonic atlas-based interrogation of fMRI data sets. *Neuroimage*, 19(3):1233–1239, 2003. 47

[205] M. Mancas and B. Gosselin. Towards an automatic tumor segmentation using iterative watersheds. *Medical Imaging*, pages 14–19, 2004. 46

[206] J. Marks, B. Andalman, P. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, et al. Design galleries: A general approach to setting parameters for computer graphics and animation. In *ACM SIGGRAPH Computer Graphics*, pages 389–400. ACM, 1997. 41

[207] H. Markus, P. Ljung, C. R. Salama, and T. Ropinski. Advanced illumination techniques for gpu-based volume raycasting, 2009. 15, 17, 23

[208] G. Marmitt, H. Friedrich, A. Kleer, I. Wald, and P. Slusallek. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proc. Vision, Modeling, and Visualization*, pages 429–435, 2004. 11, 12, 79

[209] R. Marroquim, A. Maximo, R. Farias, and C. Esperança. Volume and Isosurface Rendering with GPU-Accelerated Cell Projection. In *Computer Graphics Forum*, volume 27, pages 24–35. Amsterdam: North Holland, 1982-, 2008. 55

[210] S. Marschner and R. Lobb. An evaluation of reconstruction filters for volume rendering. In *IEEE Transactions on Visualization and Computer Graphics*, pages 100–107. IEEE Computer Society Press, 1994. 17

[211] I. MathWorks. *MATLAB: the language of technical computing. Desktop tools and development environment, version 7*. Mathworks, 2005. 10

[212] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99108, June 1995. 16

[213] J. C. Mazziotta and H. K. Huang. Thread (three-dimensional reconstruction and display) with biomedical applications in neuron ultrastructure and computerized tomography. In *AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition*, pages 241–250, New York, NY, USA, 1976. ACM. 8

[214] M. McGuire. Ambient occlusion volumes. In To Appear: *ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM, 2010. 146

[215] T. McInemey and D. Terzopoulos. Topology adaptive deformable surfaces for medical image volume segmentation. *IEEE Transactions on Medical Imaging*, 18(10):840–850, 1999. 47

[216] T. McInerney and D. Terzopoulos. Deformable models in medical image analysis. In *Mathematical Methods in Biomedical Image Analysis Workshop*, pages 171–180, 1996. 47

[217] D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Technical report, 1980. 37

[218] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, 1982. 37

[219] M. Moore and J. Wilhelms. Collision detection and response for computer animation. In *ACM SIGGRAPH Computer Graphics*, pages 289–298. ACM, 1988. 37

[220] B. Natarajan. On generating topologically consistent isosurfaces from uniform samples. *The Visual Computer*, 11(1):52–62, 1994. 11

[221] N. Neophytou, K. Mueller, K. McDonnell, W. Hong, X. Guan, H. Qin, and A. Kaufman. GPU-accelerated volume splatting with elliptical RBFs. In *Proc. Eurographics/IEEE-VGTC Symposium on Visualization*. ACM, 2006. 33

[222] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl. Cell-based first-hit ray casting. In *Proc. Symposium on Data Visualisation*, pages 77–87, 2002. 12

[223] M. Newell, R. Newell, and T. Sancha. A solution to the hidden surface problem. In *Proceedings of the ACM annual conference-Volume 1*, pages 443–450. ACM, 1972. 24

[224] NVidia. Nvidia optix. `http://www.nvidia.com/object/optix.html`, 2010. 147

[225] C. NVIDIA. Compute Unified Device Architecture Programming Guide. *Nvidia, June*, 2007. 40, 107

[226] L. Nyland, M. Harris, and J. Prins. Fast n-body simulation with CUDA. *GPU gems*, 3:677–695, 2007. 40

[227] Y. Ohtake, A. Belyaev, and G. T. M. Alexa. Multi-level partition of unity implicits. In *ACM SIGGRAPH Computer Graphics*, pages 463 – 470, 2003. 130

[228] Y. Ohtake, A. Belyaev, and H. Seidel. 3d scattered data interpolation and approximation with multilevel compactly supported RBFs. *Graphical Models*, 67:150–165, 2005. 129, 132, 134, 136, 137, 144

[229] S. Olabarriaga and A. Smeulders. Interaction in the segmentation of medical images: A survey. *Medical Image Analysis*, 5(2):127–142, 2001. 45

[230] Olegalexandrov. Level set method. `http://en.wikipedia.org/wiki/File:Level_set_method.jpg`, 2004. 46

[231] S. Osher and R. Fedkiw. Level set methods: an overview and some recent results. *Computational Physics*, 169(2):463–502, 2001. 44

[232] S. Osher and J. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Computational Physics*, 79(1):12–49, 1988. 46

[233] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Blackwell Publishing, 2007. 39

[234] M. Ozkan, B. Dawant, and R. Maciunas. Neural-network-based segmentation of multi-modal medical images: acomparative and prospective study. *IEEE Transactions on Medical Imaging*, 12(3):534–544, 1993. 47

[235] J. Painter and K. Sloan. Antialiased ray tracing by adaptive progressive refinement. In *ACM SIGGRAPH Computer Graphics*, pages 281–288. ACM, 1989. 37

[236] N. Pal and S. Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277–1294, 1993. 43

[237] M. Paliwal and U. Kumar. Neural networks and statistical techniques: A review of applications. *Expert Systems with Applications*, 36(1):2–17, 2009. 47

[238] N. Paragios. A level set approach for shape-driven segmentation and tracking of the left ventricle. *IEEE Transactions on Medical Imaging*, 22(6):773–776, 2003. 46

[239] S. Park, X. Guo, H. Shin, and H. Qin. Surface completion for shape and appearance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:168 – 180, March 2006. 130

[240] S. Parker, M. Parker, Y. Livnat, P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. In *ACM SIGGRAPH Computer Graphics*. ACM New York, NY, USA, 2005. 54

[241] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *IEEE Transactions on Visualization and Computer Graphics*, pages 233–238. IEEE Computer Society Press, 1998. 11, 12, 17, 36, 78

[242] T. Pavlidis and Y. Liow. Integrating region growing and edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 225–233, 1990. 45

[243] J. Peters, O. Ecabert, C. Meyer, H. Schramm, R. Kneser, A. Groth, and J. Weese. Automatic whole heart segmentation in static magnetic resonance image volumes. In *Medical image computing and computer-assisted intervention*, volume 10, pages 402–410. Springer, 2007. 44

[244] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. Avila, K. Raghu, R. Machiraju, and J. Lee. The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(3):16–22, 2001. 41

[245] T. Plachetka. Perfect load balancing for demand-driven parallel ray tracing. *Lecture Notes in Computer Science*, pages 410–419, 2002. 57, 89

[246] R. Pohle and K. Toennies. Segmentation of medical images using adaptive region growing. In *Proc. SPIE Medical Imaging*, volume 4322, pages 1337–46, 2001. 45

[247] S. Popov, J. Gunther, H. Seidel, and P. Slusallek. Experiences with streaming construction of SAH KD-trees. In *IEEE Symposium on Interactive Ray Tracing 2006*, pages 89–94, 2006. 37

[248] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, Sept. 2007. 28, 30, 52, 53

[249] T. Porter and T. Duff. Compositing digital images. *ACM SIGGRAPH Computer Graphics*, pages 253–259, 1984. 117

[250] M. Powell. Radial basis functions for multivariable interpolation: a review. In *Clarendon Press Institute Of Mathematics And Its Applications*, pages 143 – 167, 1987. 131

[251] M. Prastawa, E. Bullitt, N. Moon, K. Van Leemput, and G. Gerig. Automatic brain tumor segmentation by subject specific modification of atlas priors. *Academic radiology*, 10(12):1341–1348, 2003. 44

[252] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH Computer Graphics*, pages 703–712. ACM, 2002. 27

[253] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*, page 268. ACM, 2005. 27

[254] P. Rautek, S. Bruckner, and M. Groller. Interaction-Dependent Semantics for Illustrative Volume Rendering. In *Computer Graphics Forum*, volume 27, pages 847–854. Blackwell Publishing, 2008. 57

[255] E. Reinhard and F. W. Jansen. Hybrid scheduling for efficient ray tracing of complex images. In *High Performance Computing for Computer Graphics and Visualisation*, pages 78–87. Springer-Verlag, July 1995. 57, 89

[256] L. Ren, H. Pfister, and M. Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum*, volume 21, pages 461–470. Blackwell Publishing, 2002. 33

[257] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118. ACM, 2000. 31

[258] C. Rezk-Salama and A. Kolb. Opacity peeling for direct volume rendering. In *Computer Graphics Forum*, volume 25, pages 597–606. John Wiley & Sons, 2006. 57

[259] L. Robb, T. Yuen, and E. Ritman. Noninvasive Numerical Dissection And Display Of Anatomic Structure Using Computerized X-Ray Tomography. *Recent & future developments in medical imaging, August 28-29, 1978, San Diego, California*, page 10, 1978. 20

[260] S. Roettger, M. Bauer, and M. Stamminger. Spatialized transfer functions. In *Proc. of IEEE/Eurographics Symposium on Visualization (EuroVis)*, pages 271–278, 2005. 61

[261] R. Rost. *OpenGL (R) Shading Language*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004. 39

[262] S. Roth. Ray casting for modeling solids* 1. *Computer Graphics and Image Processing*, 18(2):109–144, 1982. 24

[263] S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proc. conference on Visualization*, pages 109–116. IEEE Computer Society Press, 2000. 11, 21, 32

[264] M. Rousson and N. Paragios. Shape priors for level set representations. *Computer Vision*, pages 416–418, 2002. 46

[265] S. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *ACM SIGGRAPH Computer Graphics*, pages 110–116. ACM, 1980. 35

[266] P. Sahoo, S. Soltani, and A. Wong. A survey of thresholding techniques* 1. *Computer Vision*, 41(2):233–260, 1988. 43

[267] P. Saiviroonporn, A. Robatino, J. Zahajszky, R. Kikinis, and F. Jolesz. Real-time interactive three-dimensional segmentation. *Academic Radiology*, 5(1):49–56, 1998. 97

[268] C. Salama, M. Keller, and P. Kohlmann. High-level user interfaces for transfer function design with semantics. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1021–1028, 2006. 60

[269] P. Salembier and L. Garrido. Binary partition tree as an efficient representation for imageprocessing, segmentation, and information retrieval. *IEEE Transactions on Image Processing*, 9(4):561–576, 2000. 44

[270] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984. 30

[271] H. Samet. Implementing ray tracing with octrees and neighbor finding. *Computers & Graphics*, 13(4):445–460, 1989. 30

[272] H. Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., 1990. 37

[273] D. Schlusselberg, W. Smith, and D. Woodward. Three-dimensional display of medical image volumes. In *Proceedings of the Seventh Annual Conference and Exposition, Anaheim Convention Center, Anaheim, California, May 11-15, 1986: 7th Conference: Papers.*, page 114. National Computer Graphics Association, 1986. 20

[274] J. Schulze, M. Kraus, U. Lang, and T. Ertl. Integrating pre-integration into the shear-warp algorithm. In *Proc. Eurographics/IEEE TVCG Workshop on Volume Graphics*, pages 109–118. ACM, 2003. 35

[275] J. Schwarze. Cubic and quartic roots. pages 404–407. Academic Press Professional, Inc., 1990. 11

[276] J. Sethian et al. *Level set methods and fast marching methods*, volume 18. Cambridge university press Cambridge, 1999. 46

[277] S. Shen, W. Sandham, M. Granat, and A. Sterr. MRI fuzzy segmentation of brain tissue using neighborhood attraction with neural-network optimization. *IEEE Transactions on Information Technology in Biomedicine*, 9(3):459–467, 2005. 48

[278] A. Sherbondy, M. Houston, and S. Napel. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *Proc. 14th IEEE Visualization*, pages 23 – 30. IEEE Computer Society, 2003. 39, 59

[279] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum*, volume 26, pages 395–404. Blackwell Publishing, 2007. 37, 49

[280] P. Shirley and S. Marschner. *Fundamentals of computer graphics*. AK Peters, Ltd., 2009. 11

[281] P. Shirley and R. Morley. *Realistic ray tracing*. AK Peters, Ltd., 2003. 7

[282] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Proc. Workshop on Volume Visualization*, 24(5):63–70, 1990. 32

[283] J. Sijbers, P. Scheunders, M. Verhoye, A. Van der Linden, D. Van Dyck, and E. Raman. Watershed-based segmentation of 3D MR data for volume quantization. *Magnetic Resonance Imaging*, 15(6):679–688, 1997. 46

[284] N. Simonsen and N. Thrane. A comparison of acceleration structures for GPU assisted ray tracing. *Master's thesis, University of Aarhus*, 2005. 53

[285] J. Singh and P. Narayanan. Real-time ray-tracing of implicit surfaces on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 99(1), 2009. 146

[286] P.-P. Sloan, J. Hall, J. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. In *ACM SIGGRAPH Computer Graphics*, pages 382 – 391, 2003. 135

[287] T. Song, M. Jamshidi, R. Lee, and M. Huang. A modified probabilistic neural network for partial volume segmentation in brain MR image. *IEEE Transactions on Neural Networks*, 18(5):1424–1432, 2007. 58

[288] D. Specht. Probabilistic neural networks. *Neural networks*, 3(1):109–118, 1990. 47

[289] A. Statnikov, L. Wang, and C. Aliferis. A comprehensive comparison of random forests and support vector machines for microarray-based cancer classification. *BMC bioinformatics*, 9(1):319, 2008. 59

[290] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 187–241, 2005. 13, 29

[291] K. Stevens. The visual interpretation of surface contours. *Artificial Intelligence*, 17(1-3):47–73, 1981. 8

[292] P. Sutton and C. Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In *IEEE Transactions on Visualization and Computer Graphics*, volume 99, pages 147–153. 38

[293] J. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999. 101

[294] J. Sweeney and K. Mueller. Shear-warp deluxe: The shear-warp algorithm revisited. In *Proc. symposium on Data Visualisation*, pages 95–102. Eurographics Association, 2002. 34

[295] N. Syed, H. Liu, and K. Sung. Handling concept drifts in incremental learning with support vector machines. In *Knowledge discovery and data mining*, page 321. ACM, 1999. 102

[296] I. Tobor, P. Reuter, and C. Schlick. Reconstructing multi-scale variational partition of unity implicit surfaces with attributes. *Graphical Models*, 68(1):25 – 41, 2006. 130

[297] F. Torre and M. J. Black. Robust principal component analysis for computer vision. In *Computer Vision*, volume 1, pages 59–69, 2001. 135

[298] A. Tremeau and N. Borel. A region growing and merging algorithm to color segmentation. *Pattern Recognition*, 30(7):1191–1203, 1997. 45

[299] G. Turk and J. F. O'brien. Modelling with implicit surfaces that interpolate. *ACM Trans. Graph.*, 21(4):855–873, 2002. 132

[300] F. Tzeng, E. Lum, and K. Ma. A novel interface for higher-dimensional classification of volume data. 2003. 58, 105, 106, 107

[301] F.-Y. Tzeng, E. Lum, and K.-L. Ma. An intelligent system approach to higher-dimensional classification of volume data. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):273–284, May-June 2005. 58, 105, 107, 116

[302] C. Upson and M. Keeler. V-buffer: visible volume rendering. In *ACM SIGGRAPH Computer Graphics*, pages 59 – 64. ACM, 1988. 41

[303] M. Van Kreveld, R. Van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th symposium on Computational geometry*, pages 212–220. ACM, 1997. 44

[304] M. Vannier, J. Marsh, and J. Warren. Three dimensional computer graphics for craniofacial surgical planning and evaluation. In *Proc. 10th annual conference on Computer graphics and interactive techniques*, pages 263–273. ACM, 1983. 8, 20

[305] V. Vapnik. *The nature of statistical learning theory.* Springer Verlag, 2000. 98, 101

[306] I. Viola, A. Kanitsar, and M. Groller. Importance-driven volume rendering. In *IEEE Visualization, 2004*, pages 139–145, 2004. 54

[307] I. Viola, A. Kanitsar, and M. Groller. Importance-driven feature enhancement in volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):408–418, 2005. 43

[308] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007. 39

[309] I. Wald and C. Benthin. Openrt - a flexible and scalable rendering engine for interactive 3d graphics. Technical report, 2002. 147

[310] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1):6 – 24, 2007. 39, 49

[311] I. Wald, H. Friedrich, A. Knoll, and C. Hansen. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1727–1734, 2007. 13, 38, 55

[312] I. Wald, H. Friedrich, G. Marmitt, and H.-P. Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–572, 2005. 12, 13, 28, 37, 54, 63, 65, 67, 68, 69

[313] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. 26, 27

[314] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, volume 20, pages 153–165. Blackwell Publishing, 2001. 27, 36

[315] J. Warnock. A Hidden Line Algorithm for Halftone Picture Representation., 1968. 24

[316] A. Watt. *Fundamentals of three-dimensional computer graphics*. Addison-Wesley Longman Publishing Co., Inc., 1990. 37

[317] C. Wchter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Proc. 17th Eurographics Symposium On Rendering*, pages 139–149, 2006. 38

[318] G. Weber, S. Dillard, H. Carr, V. Pascucci, and B. Hamann. Topology-Controlled Volume Rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2):330–341, 2007. 62

[319] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 44. IEEE Computer Society, 2003. 11

[320] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *ACM SIGGRAPH Computer Graphics*, volume 32, pages 169–179. ACM, 1998. 11, 31, 36

[321] R. Westermann and B. Sevenich. Accelerated volume ray-casting using texture mapping. In *Proc. 1st IEEE Visualization*, pages 271–278. IEEE Computer Society, 2001. 11, 27

[322] L. Westover. Interactive volume rendering. In *Proc. Chapel Hill workshop on Volume visualization*, pages 9–16. ACM, 1989. 33

[323] L. Westover. Splatting: a parallel, feed-forward volume rendering algorithm. Technical report, 1991. 33

[324] T. Whitted. An improved illumination model for shaded display. 1980. 21, 24

[325] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201 – 227, 1992. 10, 37, 38

[326] J. Wilhelms and A. Van Gelder. A coherent projection approach for direct volume rendering. Technical report, 1991. 27, 32

[327] A. Williams, S. Barrus, R. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. *Journal of graphics tools*, 10(1):49–54, 2005. 75

[328] L. Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274, 1978. 24

[329] T. Williams. *A man-machine interface for interpreting electron density maps.* PhD thesis, 1982. 8

[330] W. V. Wright. *An interactive computer graphic system for molecular studies.* PhD thesis, 1972. 8

[331] H. Xie, K. T. McDonnell, and H. Qin. Surface reconstruction of noisy and defective data sets. In *IEEE Visualization*, pages 259 – 266, 2004. 130

[332] R. Yagel and A. Kaufman. Template-based volume viewing. In *Computer Graphics Forum*, volume 11, pages 153–167. Blackwell Publishing, 1992. 27

[333] R. Yagel, A. Kaufman, and Q. Zhang. Realistic volume imaging. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 226–231, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. 7

[334] S. Yoon, S. Curtis, and D. Manocha. Ray tracing dynamic scenes using selective restructuring. In *Proc. of Eurographics Symposium on Rendering*, pages 73–84, 2007. 39

[335] R. Zawadzki, A. Fuller, D. Wiley, B. Hamann, S. Choi, and J. Werner. Adaptation of a support vector machine algorithm for segmentation and visualization of retinal structures in volumetric optical coherence tomography data sets. *Biomedical Optics*, 12:041206, 2007. 59

[336] D. Zhang and S. Chen. A novel kernelized fuzzy c-means algorithm with application in medical image segmentation. *Artificial Intelligence in Medicine*, 32(1):37–50, 2004. 47

[337] Y. Zhang. A survey on evaluation methods for image segmentation* 1. *Pattern Recognition*, 29(8):1335–1346, 1996. 43

[338] Y. Zheng, A. Barbu, B. Georgescu, M. Scheuering, and D. Comaniciu. Fast automatic heart chamber segmentation from 3D CT data using marginal space learning and steerable features. In *Proc. ICCV*, volume 18, pages 31–94. Citeseer, 2007. 44

[339] J. Zhou, A. Doring, and K. Tonnies. Distance based enhancement for focal region based volume rendering. *Proceedings of Bildverarbeitung fur die Medizin04*, pages 199–203, 2004. 56, 60

[340] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia*, pages 1–11. ACM, 2008. 37, 54

[341] S. Zhu, T. Lee, and A. Yuille. Region competition: unifying snakes, region growing, energy/Bayes/MDL for multi-band image segmentation. In *iccv*, page 416, June 1995. 45

[342] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6(1):1–18, 1988. 40

[343] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA volume splatting. In *Proc. conference on Visualization*, pages 29–36. IEEE Computer Society, 2001. 33