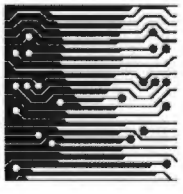
INCREMENTAL VOLUME RENDERING USING HIERARCHICAL COMPRESSION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Michael B. Haley
1 May 1996

Supervised by
Prof. Edwin H. Blake



Abstract

The research has been based on the thesis that efficient volume rendering of datasets, contained on the Internet, can be achieved on average personal workstations. We present a new algorithm here for efficient incremental rendering of volumetric datasets. The primary goal of this algorithm is to give average workstations the ability to efficiently render volume data received over relatively low bandwidth network links in such a way that rapid user feedback is maintained. Common limitations of workstation rendering of volume data include: large memory overheads, the requirement of expensive rendering hardware, and high speed processing ability. The rendering algorithm presented here overcomes these problems by making use of the efficient Shear-Warp Factorisation method which does not require specialised graphics hardware. However the original Shear-Warp algorithm suffers from a high memory overhead and does not provide for incremental rendering which is required should rapid user feedback be maintained. Our algorithm represents the volumetric data using a hierarchical data structure which provides for the incremental classification and rendering of volume data. This exploits the multiscale nature of the octree data structure. The algorithm reduces the memory footprint of the original Shear-Warp Factorisation algorithm by a factor of more than two, while maintaining good rendering performance. These factors make our octree algorithm more suitable for implementation on average desktop workstations for the purposes of interactive exploration of volume models over a network. This dissertation covers the theory and practice of developing the octree based Shear-Warp algorithms, and then presents the results of extensive empirical testing. The results, using typical volume datasets, demonstrate the ability of the algorithm to achieve high rendering rates for both incremental rendering and standard rendering while reducing the runtime memory requirements.

Acknowledgements

I would like to extend my gratitude to my supervisor Prof. Edwin Blake for his guidance and advice throughout my research. Thanks must also go to my fellow students and friends whose support was invaluable.

I would like to acknowledge the University of North Carolina (Chapel Hill) for making some of the volumetric test data freely available. Recognition should also be extended to Stanford University for making the Shear-Warp algorithm and test data available.

I also acknowledge the support of the Foundation for Research Development (FRD) for funding the first year of this research.

Contents

Chapter 1

Introduction

1.1 Aims and Contributions

The primary aim of this research was to develop a more effective method for rendering volume data on average desktop workstations as well as providing an efficient means for distributing volume data over a conventional network.

We found that through a unique combination of compression techniques and rendering algorithms, these aims would be realised. The final *package* of algorithms presented in this dissertation provides for the rapid transmission of volume data (in a compressed form) over a network to an average desktop workstation which can then render this data incrementally (i.e. as it arrives) in its compressed form. Once the data is completely transmitted, any further manipulations of the volume model (e.g. rotations, scales, translations) can be performed in (typically) under 2 seconds and once again only using the compressed data structures.

1.2 Volume Visualisation

Volume Visualisation is the science of rendering images of three dimensional datasets. Generally these datasets consist of three dimensional arrays of either scalar values or vectors. We will only be considering the scalar volumes. These datasets are generated (and require visualisation) by numerous diverse disciplines.

With the advent of superior medical imaging techniques such as Magnetic Resonance (MR) and Computed Tomography (CT) scanning, highly detailed images of "slices" through the human body may be captured. Historically these images were treated, purely, in their two dimensional form for the purposes of diagnosis. This limitation was primarily due to the memory required for storing the images and the processing required for manipulating them. According to Moore's law [1], processing performance doubles every 18 months and a similar (but slower) effect is seen with primary and secondary storage. With growth like this it has rapidly become feasible to generate three dimensional models of areas of the human body directly from the images obtained using the methods mentioned

above. These three dimensional models are however extremely memory intensive and, historically, require large amounts of processing power to generate images of the models.

The advantages of these models are great. With a comprehensive model of a region of a patient's body a surgeon may practise or plan certain operations [2] without touching the patient. Also once the model is captured and constructed multiple different views of areas of the body may be generated unlike previous two dimensional imaging where the images had to be directly captured. This increases the chances of a valid diagnosis and a successful operation.

Volume visualisation has not however been confined to the medical arena. Another impact of powerful computing has been the implementation of complex numerical simulations such as fluid flow dynamics or finite element models. These simulations generate very complex three dimensional (or sometimes higher) data which is too complex to understand in its numerical form. This data thus has to visualised in order for it to be understood.

In the late 1970's and 1980's the only methods of visualising volume data consisted of very time consuming polygon mesh construction. This process generated a polygonal approximation of an isosurface within the volume, where the isosurface value was specified by the user. The polygonal model was then rendered using classical rendering methods such as scan-conversion or ray-tracing. These methods were fairly slow and were found to be very inflexible as a new polygon approximation of the volume needed to be computed every time a new region of the volume was to be visualised.

In the late 1980's and early 1990's new methods of directly rendering the volume data began to appear. These methods traditionally used ray-tracing to cast rays through the volume, which was treated as a region of very small cubes each with its own opacity and shading. Although this method generated very realistic images the rendering times were still too long for meaningful use. Another approach to directly rendering volumes was the compositing approach, where the image plane was incrementally passed through the volume and the effects of each small shaded cube on the image plane was taken into account. This dramatically accelerated the volume rendering process.

In recent years the compositing approach has been improved to the point that images, of a standard sized volume, may be rendered in under 1 second on an average high-end workstation. As with most algorithms there is a trade-off between memory size requirements and processor speed requirements. The memory requirements of these volume rendering algorithms are still very high making them unsuitable for average *desktop* workstations. According to Moore's law (as mentioned above) processor speed is increasing at an exponential rate, while memory capacity is also increasing but at a slower rate. This indicates that to achieve rapid rendering of volume data on a desktop workstation the algorithms should be adapted to put higher requirements on processor speed but lower requirements on memory capacity.

This dissertation will present a new modification to the method of data storage used during composite volume rendering which involves a compressed data structure and caching techniques. This modification will greatly reduce the primary memory required to render volumes without impacting the

performance of the renderer. By reducing the primary memory usage the rapidly increasing speed of microprocessors will make the rendering of standard volumes on any average desktop workstation a reality.

Another massive impact that rapidly growing technology (and acceptance of it) has resulted in is the inter-networking of computers. The inter-networking of workstations opens up many new possibilities in the various fields of visualisation, such as distributed computing, collaboration during visualisation, and remote diagnosis.

1.3 Hypermedia and the World-Wide-Web

In recent years the world wide Internet network of computers has gained mainstream acceptance. (A January 1996 survey of the Internet reported over 9 million hosts on the Internet.) With this mainstream acceptance has come increased band-width connections and a wide range of facilities and information. The economics of this acceptance has resulted in very cheap network connection options resulting in even more people connecting to the Internet.

Hypermedia allows users to naturally follow references in data and thus obtain necessary information in a comfortable (by this I mean representing the data in the most natural form possible, such as video, audio, etc.) and efficient manner. The most widely accepted form of hypermedia in the world is now the World-Wide-Web (WWW). A recent report demonstrated that one out of every 270 computers on the Internet was a WWW server, providing hypermedia documents. (This represented a 50% growth in 6 months!) As mentioned above, a hypermedia system should provide for a wide range of media forms in the hypermedia documents. The WWW primarily uses text and two-dimensional images. Usage of sound and video clips is also common.

Recently a proposal of a virtual reality extension to the WWW was made and accepted. This extension is called the Virtual Reality Modelling Language (VRML) and it stems primarily from *Silicon Graphic's OpenInventor* scripting language for describing polyhedral models. VRML (see [3] for the specification) allows three dimensional polygonal models to be transmitted over the Internet to a WWW browser application. This browser application (when recognising VRML data) will render the three dimensional model for the user. The user may then interact with the model by moving a camera (the user's view point) around the small three-space environment. Various polygonal models in the VRML environment may be associated with other WWW documents (perhaps other VRML data) so that references may even be followed in this virtual space. Extensions to VRML now include object behaviour description (using the *Java* language) and Universal Resource Names (URNs) for identifying certain common objects.

Unfortunately the VRML specification does not cater for volume data (as mentioned in §1.1) but restricts itself to polygonal models for the sake of performance. This decision is understandable considering that mostly average desktop workstations will be receiving and rendering these models and rapid user feedback is necessary in any visualisation system.

This dissertation proposes a method for transmitting compressed volume data over the Internet such that rapid user feedback is maintained. The algorithms developed will also be able to execute efficiently on average desktop workstations, thus opening up the possibility of making three dimensional volumetric data available on the WWW.

Making this data available on the WWW opens up many interesting visualisation opportunities ranging from collaboration to remote diagnosis.

1.4 Distributed and Collaborative Visualisation

Due to the complexities of certain forms of data it becomes necessary to distribute the computation over numerous computers on a network, and in some cases to allow multiple people to access and visualise the data simultaneously. In the case of medical volume data captured from MR or CT scanning (mentioned above) the possibility of remote diagnosis arises. Should a patient's body be scanned at a certain location in the world, where a specialist diagnosis of a particular problem is not available, numerous specialists with access to a computer anywhere in the world could access the data and perform diagnoses.

The visualisation of any form of data generally implies a number of stages during which the data is filtered and eventually rendered into an image which is meaningful to the user. Depending on the complexities of the data these stages might vary greatly in the length of time required to execute. (e.g. It may be very easy to filter some data forms but the rendering might be very time consuming, while a different type of data may be difficult to filter but may render rapidly.) Also the amount of data generated between stages might vary from a few bytes of information to gigabytes of information. It can therefore become necessary to distribute these visualisation stages over a network of computers where different computers handle certain operations best suited to them (perhaps in parallel with other operations being performed on other computers).

With volume visualisation it is necessary to perform numerous processing operations on extremely large datasets for the final rendered image. Depending on the method of user interaction with the volume model some of this processing may need to be repeated. Considering that the volume data is very large it is desirable to, firstly, transmit data between distributed computers only when the data is in its smallest state. Secondly, the distribution should be such that a small (and fairly common) operation by the user will not result in re-transmission of data constantly between the distributed computers. In summary the primary idea here is to attain rapid user feedback by intelligently distributing the stages of the visualisation process between a number of computers.

Many of the network links which support the Internet are not high bandwidth links, thus the transmission of large volume datasets could become time consuming. Should procedures such as remote diagnosis become useful then the volume visualisation stages will have to split in such a way that the data transmitted to the client is as small as possible and that constant transmission is not required during user manipulations. This implies the use of rapid compression/decompression techniques for the transmission of the volume data.

The algorithms presented in this dissertation address these challenges and provide a method of pre-processing the volume data on a server and then incrementally transmitting this data to a client (in a compressed form). The client may then render images of an approximation of the volume directly from the compressed data thus not incurring a large decompression overhead. This gives the user an initial impression of the volume model and allows the setting of various viewing parameters. As the volume data is transmitted this approximation improves until, eventually, the volume itself is rendered exactly. The split of visualisation stages is such that manipulation of the volume does not require any retransmission of data.

Users should respond favourably to having rapid overviews of the volume and the ability to adjust the parameters during volume transmission. Generally a user will find such a network based visualisation system acceptable if user-feedback is maintained throughout the visualisation process irrespective of network band-width. Our algorithm achieves both these goals.

1.5 Overview

The remainder of this dissertation is presented in the following fashion:

- *Chapter.2 - Background*: Presents detailed background information on the development of volume rendering techniques and the associated theory. The chapter leads up to the latest developments in direct volume rendering and ends by outlining how the algorithms presented here relate to previous work.
- *Chapter.3 - Octree Compression and Data Classification*: Presents the octree compression technique used to represent the volume data in all the algorithms. The methods and theory of the compression algorithm are presented and incremental transmission is discussed. The reception of the incremental volume data is then presented and algorithms for incremental data classification are described.
- *Chapter.4 - Rendering Hierarchical Volumes*: Details the algorithm for rendering the octree compressed volume data as well as the underlying mathematical theory for the Shear-Warp algorithm. The methods for approximating the volume for incremental rendering are also discussed.
- *Chapter.5 - Experimental Results*: The basic hypotheses of the thesis are presented and methods for testing them are discussed. The testing methodology is detailed and then results for all the volume processing algorithms are presented.
- *Conclusion*: An overview of the dissertation indicating areas where significant solutions are proposed and areas where problems still exist. Future directions in this research are also outlined.
- *Appendix.A - Glossary*: A glossary of common technical terms used throughout the dissertation.

- *Appendix.B - The VOX++ class library*: A description of the C++ class library which implements all the algorithms presented here. The library also provides an extensible framework for testing volume rendering algorithms. A full reference of every object is also provided.
- *Appendix.C - Graphs*: Most of the detailed results of the tests discussed in *Chapter.5* are presented in graph form.
- *Appendix.D - Images*: Visual results from the rendering algorithms Comparisons are made between the traditional Shear-Warp algorithm and the octree algorithm, using a variety of rendering parameters.

Chapter 2

Background

2.1 Introduction

In the field of data visualisation the data may be in various formats, the most common of which are:

- $V[t]$ Vectors of length t containing scalars. (e.g. Simple statistical tables.)
- $S[t]^2$ Two dimensional arrays of scalar data. (e.g. Monochromatic Images.)
- $V[t]^2$ Two dimensional arrays of vector data. (e.g. Colour Images, or surface flux data.)
- $S[t]^3$ Three dimensional arrays of scalar data. (e.g. Medical volumes scanned by CT or MR)
- $V[t]^3$ Three dimensional arrays of vector data. (e.g. Multimodality volumes, fluid-flow data.)
- $S[t]^N$ N-dimensional arrays of scalar data. (e.g. Abstract simulation data or time varying volumes)
- $V[t]^N$ N-dimensional arrays of vector data. (e.g. Abstract simulation data.)

The field of standard volume visualisation which we will concern ourselves with in this thesis falls into the $S[t]^3$ category (i.e. A three-dimensional array of scalars).

The field of volume visualisation came about from the merging of medical image processing and solid modelling and has been an active field of research for the last 10 years. Historically, traditional medical imaging concerned itself with two dimensional images while solid modelling dealt solely with polyhedral solids using methods such as scan conversion, ray-tracing, or radiosity. With emergence of higher resolution medical scanners, larger primary and secondary storage, and better graphics and processing technology, the concept of three dimensionally rendering solid volumes of image data emerged. Of course this field is not at all limited to medical uses anymore, and a lot of work is being performed in areas such as fluid-dynamics simulations and finite-element models.

This chapter will outline the exact nature of the problem, and will present some of the previous work performed in this field.

2.2 Distributed Visualisation

Over recent years network bandwidths have increased sufficiently for many distributed network approaches to visualisation to be realised.

Anupam et al. [4] have developed the SHASTRA environment for collaborative multimedia scientific manipulation. The authors have implemented a volume rendering system which distributes the rendering calculations between rendering servers based on the complexity of rendering areas of the final image. Their environment also supports collaborative viewing so that a number of users may explore the same dataset from different viewpoints and interact with one another. The users can collaborate in the setting of viewing parameters and can inspect the results independently.

Recently Law & Yagel [5] presented a highly efficient distributed volume rendering system which makes use of an advancing ray-front (a block of simultaneously advancing rays) to achieve greater coherency in the data access during rendering. This greater coherency then better exploits their local caches.

Simon et al. [6] present the *Multimedia MedNet* used by a number of hospitals and research laboratories. Their system does not provide for volume rendering but merely the efficient distribution of multimedia data over a wide-area-network (WAN). The system allows for data being captured during surgery (video. audio, neurophysiological, and autonomic data) to be simultaneously distributed over the network to numerous collaborative users. This allows experts to examine and comment during the actual surgical operations.

One of the primary advantages of this trend towards distribution and collaboration as far as medical data goes, is that experts in various fields may be consulted with ease. Also due to the inception of high-bandwidth network links there is no real limitation on the location of the expert either.

With the explosion in popularity of the Internet and the World-Wide-Web hyper-media browsing system a lot of necessary data is being made widely available. Unfortunately many of the network links over which the Internet is propagated are relatively low bandwidth, and the transferral of large amounts of data (e.g. real-time video) is still not possible. However on the other hand most people in professional organisations and research institutions now have easy (and cheap) access to the Internet, so it makes sense to move the collaborative systems mentioned above to the Internet.

Ang, Martin, and Doyle [7] have presented a mechanism for controlling volume visualisation over the World-Wide-Web (WWW). Their system adds controls to a user's WWW browser which send messages back to a volume rendering server. This server then renders images of the volume with the requested parameters and transmits these images back to the user. This technique, while being operative, is naïve in that: (a) simultaneous usage by many users will result in very poor performance, and, (b) transmitted images will not arrive at the user in a reasonable amount of time.

In order to distribute volume models over the Internet it is necessary to properly distribute the load of the volume rendering process. With the advent of rapid new volume rendering techniques (which no longer require high-end workstation performance) the rendering may be performed on the user's own

workstation. In this case efficient means of compressing the volume for transmission is required. Also due to the low-bandwidth of many Internet links it could take in the order of 20 minutes to transfer an average sized volume to a user. It would thus be advisable to render the data (rendering times are in the order of seconds) incrementally during its arrival. This would maintain a reasonable user-feedback. The algorithms presented in this dissertation achieve this goal.

2.3 Volume Representations and Compression Techniques

The question of how to represent and store a volume of data, depends very much on the nature of the original data and on the methods which will be used to visualise this data.

2.3.1 Nature of the data

Volumetric data is usually represented as a three-dimensional lattice where either the vertices of the lattice or the areas bounded by connecting planar vertices are voxels. (A voxel is a three dimensional data element, much like the three dimensional equivalent of a pixel.) This lattice may then take one of two forms:

- Regular - The spacing of vertices is constant on any one axis.
- Irregular - The spacing of vertices is not constant.

Most medical datasets and other datasets resulting from direct scanning by CT or MR methods are regular lattices of data. Data produced from numerical simulations or models produced for finite element modelling are generally irregular lattices of data [8,9]. In this thesis we will only be considering regular data. It is normally possible to convert irregular data to regular data by performing a resampling operation on the data using some form of reconstruction filter. However the reconstructed regular dataset may be very large and have undesirable aliasing artifacts in it.

Computed Tomography (CT) and Magnetic Resonance (MR) scanning both use a process of tomography to construct two dimensional slices through the object being scanned. Then by moving the object in a direction orthogonal to the slices, numerous aligned slices can be scanned at set points through the object. Once these slices (which essentially form a number of two dimensional images) are captured they may then be "stacked" together to form a three dimensional volume of data. The distance between neighbouring voxels (or lattice vertices) may not be the same in every direction and this information should be stored with the volume to allow for accurate reconstruction. The resulting data, in its raw form, takes the form of a three dimensional array of integers. References to this data format can be found in [10].

The size of these datasets, however cause many problems ranging from simple storage space to long rendering times. Considering an average medical dataset could be of the size 256x256x100 with every voxel represented by a 2-byte integer, this results in 13 107 200 bytes of data, and during the visualisation of the data more information may need to be calculated on a per-voxel basis further increasing this size.

Performance problems are invariably encountered when loading the data from secondary storage or transmitting the data over a network link. Also in most virtual memory based operating systems, allocations of very large data structures which are constantly accessed in varying orders can cause the operating system to "thrash" (constantly swap memory between primary and secondary storage), thereby dropping the performance of the entire workstation. To address these problems researchers have developed a range of volume compression schemes to reduce the size of the data and to filter out unnecessary data. These schemes will be covered in §2.3.2, §2.3.3, §2.3.4, and §2.4.

The rendering of large volumes also presents a major problem due to the amount of data which has to be referenced during the rendering process. This problem arises for most rendering methods ranging from isosurface rendering to volume ray-tracing. (These rendering methods are covered further in §2.5.) Other rendering problems include: generating alias-free images, locating different materials in the volume more accurately, and partial rendering of the volume (i.e. As a overview or during rapid animation). These problems have led to the development of a variety of volume representation schemes, each one implying a different approach to the rendering.

2.3.2 Pyramid Representation

Pyramid representation consists of recursively subdividing the volume into a full tree (or pyramid) of hierarchically organised voxels. Figure 2.1 depicts the pyramid structure for a 4x4x4 volume. Often this pyramid of nodes is trimmed so that certain branches are discarded. The resultant data structure is referred to as an *octree*.

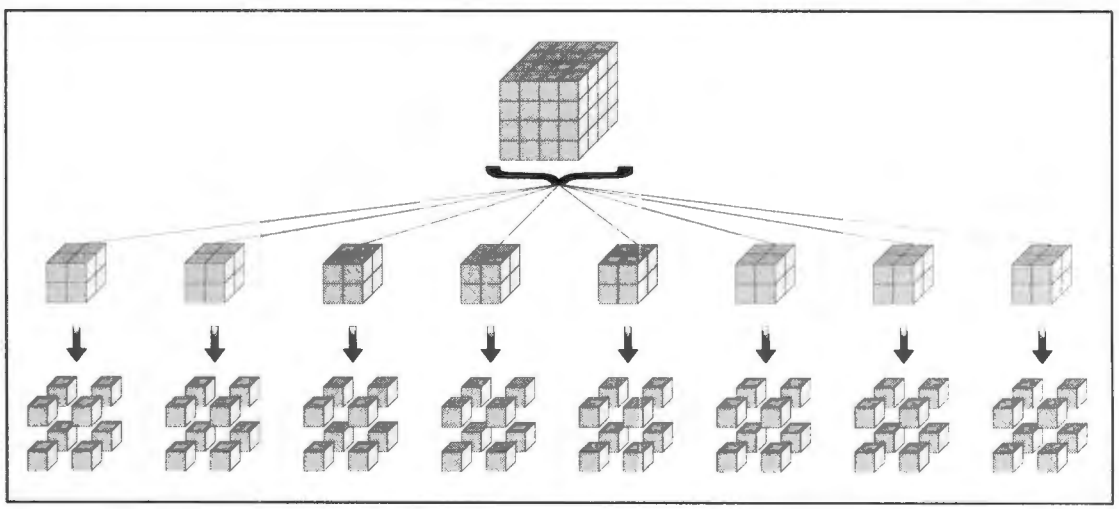


Figure 2.1 - Pyramid reduction of a 4x4x4 volume dataset.

Wilhelms and Van Gelder [11] discuss the use of octrees for the rapid calculation of isosurfaces. They use a specific type of octree called a branch-on-need octree in conjunction with a node cache to efficiently determine the isosurface of a volume.

A novel use of the pyramid representation was presented by Laur and Hanrahan [12]. They use the generalising ability of non-leaf octree nodes to generate approximations of areas of the volumes.

These approximations are then rendered as textured *splats* (the compositing of an area of pixels into the final image) using hardware Gourard shading facilities. This technique allows the authors to rapidly render the volume when detail is unnecessary, for example, during rapid animation.

Levoy [13] proposed the use of an octree data representation to rapidly omit transparent areas of the volume during the rendering process. This served to decrease the amount of per-ray calculations during the ray-tracing process which he used.

2.3.3 Frequency Domain Representation

The Fourier *Projection-Slice Theorem* allows 2 dimensional projections to be generated from a three dimensional dataset in frequency space. In order to achieve this, the original volume is transformed into a three dimensional frequency space representation. Unfortunately the usage of frequency space entails an increase in the overall size of the volume as well as incurring the overhead of Inverse-Fourier Transforming the slice for every rendering. For more information see the section on frequency domain rendering below. (§2.5.5)

2.3.4 Multiresolution Representation

Multiresolution techniques strive to represent the volume in such a way that there is a hierarchical decomposition of the volume into a number of geometric approximations. These approximations should optimally tend towards an exact replica of the original volume, as the resolution increases. Using this approach: run-time memory usage during rendering may be reduced, overall approximation volumes may be rendered rapidly, and by exploiting the nature of the human eye certain features of the volume may be filtered out to reduce the overall volume size.

Muraki [14] originally proposed the use of Blinn's *blobby* model representation of surface, to achieve a multiresolution representation of a volume's isosurface. While being successful this method required high amount of approximation volumes (or blobs) to achieve a visually recognisable picture. The method was also extremely slow, requiring days to render a single image.

With the advent of Wavelet Theory [15], a whole new approach to multiresolution representation became possible. Westermann [16] gives a good theoretical overview of the wavelet decomposition method with respect to the volume rendering integral. He compared the use of three different basis functions for the three dimensional wavelet decomposition, and found that the Daubechies wavelet basis produces better approximations, but the Haar basis allows for faster rendering. The method reduces the overall memory requirements drastically. However the images obtained were of a fairly low quality and the rendering times were in the order of 10 minutes.

Muraki [17] also presented the use of a three dimensional wavelet transform for producing a multiresolution volume. He made use of the Battle-Lemarié wavelet basis which gives results similar to his earlier *blobby* model work. While the transformation time was in the order of minutes, the rendering times were found to be in excess of 10 hours.

A much better wavelet basis was later proposed by Muraki [18]. He used the three dimensional difference of Gaussians (DoG wavelet) functions as the wavelet basis. This gave results close to that produced by the *meta-ball* surface approximation method used extensively in solid modelling and ray tracing. The rendering times were now reduced to under 20 minutes.

Further work in the field of wavelet decomposition of volumes primarily consists of trying to find a better set of basis functions, which may be rendered efficiently while representing the volume accurately. Guo [19] has recently proposed the use of wavelet maxima and alpha shapes for the reproduction of volume surfaces, however the method cannot represent standard medical datasets well.

A different approach to the multiresolution representation was presented by Ranjan and Fournier [20]. They propose the use of unions of spheres for the representation of isosurfaces. Their method while achieving reasonable rendering rates (30 seconds) cannot represent the whole discrete volume and is limited to isosurfaces only.

On reflection, the *Pyramid Representation* section presented a number of octree methods which could also be considered to be multiresolution representations. Due to the efficient nature of constructing and maintaining octrees, this method hints at a very efficient rendering scheme for octree representations [21].

2.4 Compression Techniques

Compression of the volume data is a fairly common approach and ranges from *lossless* techniques, which preserve all the information content of the volume, to *lossy* techniques which approximately reconstruct the original volume on decompression.

2.4.1 Lossless compression

A lossless compression function is a function which accepts an input data stream and generates a smaller data stream which, when operated on by the decompression function, regenerates the original data stream exactly. Thus the redundancy in the original data stream is exploited.

One of the oldest but sometimes most effective methods (normally in terms of processing speed and ease of implementation) is the *run length encoding* method where runs of similar values are coded into a representative of that value as well as a count of the recurring values. Montani and Scopigno [22] developed a volume representation and rendering scheme called STICKS, where the volumes of data were compressed using run length encoding on the z-axis runs (or sticks) of data. Their algorithm also made use of a lookup table to quickly locate the start of encoded sticks, as well as to allow secondary storage of some of the data in a scheme similar to virtual memory.

A promising extension to the STICKS method was proposed by Shareef and Yagel [23] using a data structure called a segment wall. Their data structure contains segments of voxels which contain either empty runs or runs of equivalently valued voxels. They also develop an efficient beam-tracer which is used to render this data and they demonstrate its use by applying it to a CSG (constructive solid geometry) modelling system. More recent algorithms such as the *Shear-Warp* algorithm by Lacroute

and Levoy [24,25] exploit the nature of the run length encoded data to achieve very high rendering speeds.

Another approach presented by Fowler and Yagel [26] is the combination of a differential pulse-code modulation and Huffman coding. In this method a combination of a predictor and a Huffman coder are used to achieve a near-optimal lossless compression of the volume. However the compression and decompression times are in the order of minutes and the data structures do not allow random accessing of the compressed data. This therefore excludes the possibility of directly rendering the data in its compressed format.

2.4.2 Lossy compression

A lossy compression function is a function which accepts an input data stream and generates a smaller data stream which, when operated on by the decompression function, regenerates the original data stream to within a specified degree of accuracy. This degree of accuracy is specified during the compression phase and is referred to as a *quality factor*. This indicates how much of the original data is to be maintained. These compression algorithms generally achieve very high compression ratios.

Many of the compression techniques used in volume compression are merely extensions of two dimensional techniques used for image compression. This is certainly true for many of the lossy volume compression algorithms.

Ning and Hesselink [27] present a technique based on vector quantization. Here the original volume is converted into a set of indices which reference a code-book of data vectors. The quality factor effects the size of the code-book. Their technique also allows for the rendering of the data in its compressed form. This is very advantageous as a separate decompression overhead is not incurred.

Yeo and Liu [28] propose the use of a three dimensional discrete cosine transform (DCT) similar to that used in the JPEG image compression scheme. Their method (like JPEG) breaks the volume up into 8x8x8 blocks each of which is transformed using the DCT, quantised (using a pre-set table), and Huffman coded. Edge of block artifacts are reduced using overlapping *macro* blocks of size 32x32x32. Their method achieves compression ratios of between 20 and 30 percent without generating noticeable artifacts. They also proposed that the decompression is performed during the rendering process which results in fairly long rendering times.

It is interesting to note that this DCT compression scheme could fit elegantly into the octree (or pyramid) representation scheme discussed in the previous section. This is due to the *power of 2* size of the blocks and macro-blocks and the equivalent size of octree nodes.

2.4.3 Consequences

The use of *lossy* compression is often not acceptable, especially in the fields of medical diagnosis. This is due to the generation of compression *artifacts* brought about due to the inexact representation. These artifacts could be perceived as false objects in the final rendered image, thus allowing the possibility of incorrect diagnosis.

The rendering algorithm which is presented in this dissertation, while allowing for the incorporation of lossy compression, chooses to use only lossless compression. This choice is motivated by the above artifacts problem, as well as the potential time delay incurred by the more complex decompression process implied by many of the lossy compression algorithms.

2.5 Volume Rendering

2.5.1 Background

The discrete volume of voxels described in the previous sections can be rendered in any orientation with particular scalings and translations. Numerous coordinate systems are used during the rendering process. The most notable are:

- **Object Coordinates** - The coordinates of voxels in the discrete three dimensional array.
- **World Coordinates** - The coordinates of the volume in 3-space as perceived by the viewers "camera". This includes all transformations produced on the volume.
- **Image Coordinates** - The coordinates of the pixels in the final rendered image. (These coordinates also have analogies in world coordinates.)

For some rendering methods an isosurface or cuberilles is used. With these methods a particular density level (or closed range of levels) in the volume is selected for rendering. All of the voxels at this particular density level will define some form of three dimensional shape.

Using the cuberilles method all voxels which are not at the specified density level are removed and only the voxels remaining are rendered as a collection of small cubes in space. This rendering may be performed by any multi-purpose solid modelling method such as Z-buffered scan conversion or ray-tracing.

The isosurface method consists of locating all voxels within the specified density range and interpolating them together to form a number of continuous surfaces. These surfaces are generally approximated by large numbers of polygons using algorithms such as the Marching Cubes algorithm [29]. These polygons are then also rendered using the same methods as for cuberilles.

Another prominent method of volume rendering is direct volume rendering where the volume is taken be a varying semi-transparent medium[30,31]. When rendering, light rays are traced through the volume and projected onto an image plane in the same way that ray-tracing is performed. As the light ray passes through the volume it is attenuated and reflected. The basic equation for the intensity of the resulting ray is given by:

$$I(t_0,t_1) = \int_{t=t_0}^{t_1} q(t)e^{-\int_{s=t_0}^{t}\sigma(s)ds}dt$$

where $\sigma(s)$ defines the attenuation function of the material in the volume, $q(t)$ is the volume intensity at position t along the ray, and t_0 and t_1 are the entry and exit points of the ray. Most direct volume rendering methods can be considered to be approximations of this integral.

Normally before the volume is rendered a number of filtering operations have to be performed. Also a large amount the calculations required during rendering can be pre-computed and stored with each voxel. This process is known as classification.

2.5.2 Pre-processing & Classification

The pre-processing of a volume can consist of the following stages:

- The construction of the discrete three dimensional array from a particular medical data format such as DICOM or ACR-NEMA.
- The range correction of density values as the volume may not have optimal *contrast*.
- Re-sizing of the volume using signal resampling [10]. Here the original three dimensional signal of the volume is reconstructed (using Fourier analysis, say). Once this signal is established it is then re-sampled to create a different size volume. Of course the Nyquist limit has to be adhered to otherwise severe aliasing artifacts will appear.
- Some scanning methods are very susceptible to noise at certain locations in the volume. The three dimensional analogy of many image processing noise filters can be applied to remove the noise provided that the statistical characteristics of the noise is understood [30].

The classification stage generally consists of the following operations:

- Assignment of a shading and opacity to each voxel.
- Calculation of a surface normal and gradient magnitude at each voxel.
- Removal of transparent voxels.

The assignment of shading and opacity is generally performed by a collection of material definitions as well as an opacity transfer function. The material definitions, define the colour and shading characteristics (ambient, diffuse, and specular shading). The mapping of density values to material definitions can be many-to-one or many-to-many by allowing fractional material contributions to certain voxel densities. The opacity transfer function, which generates an opacity, is normally a function of voxel value and gradient magnitude.

The surface normal and gradient magnitude is calculated using a finite differences method operating on the neighbours of a voxel. The approximation of the normal may be more accurate by using a larger number of neighbours but this is often not desirable as more data has to be stored on a per-voxel basis.

To optimise the rendering process of the volume the voxels which will be completely transparent, as a result of the transfer function, can be eliminated. Generally this comprises a fairly large percentage

(70% or more) of the volume and provided an efficient data structure is used (such as an octree) the rendering will be vastly accelerated.

Sobierajski et al. [32] propose a data structure which contains only the outer most voxels visible voxels (called a *trimmed voxel list*) thus reducing the number of voxels sent through to their geometry engine. This method is very similar (but more efficient) to Udupa and Odhner's [33] *semi-boundary* data structure. Both of these methods do however require some display time computations of a modified data structure. A more recent approach by Yagel et al. [34] uses a data structure called a *fuzzy voxel set* which consists of voxels which contribute significantly to the final image. (This contribution is computed by measuring the effect of splatting a single voxel has in relation to the entire image.)

2.5.3 Multimodality data

A recent advance in volume visualisation is the use of multimodal data. This means that data captured through a number of different methods is combined together to achieve very informative renderings of the volume. In the medical field CT scanning produces very good bone definition, while MR scanning produces good tissue definition. Also scanning such as SPECT can produce maps of the electrical activity in the brain.

Zuiderveld [35] developed an object oriented rendering system which incorporates multimodal data by using many of the implicit advantages of ray tracing. A large section of his method deals with the mapping of the various data types together. Also the process of registration (or alignment with each other) of the datasets is still a field of active research.

Recently the National Library of Medicine's *Visible Human Project* was completed, where numerous scans of a male and a female cadaver were collated. The scans consisted of

- An MR scan of the entire cadaver.
- A CT scan of the entire cadaver.
- A physical slicing of the frozen cadaver into millimetre thick slices, followed by the imaging of these slices.

These various modalities were registered together and are now being made available to the scientific community. Tiede, Schiemann, and Höhne [36] present some of the first renderings of this data set, taking full advantage of the multimodality nature of the data. Due to the fact that actual RGB colour data is available for the voxels, highly realistic images can be rendered.

2.5.4 Isosurface methods

As mentioned in the *Background* subsection above, isosurfaces are a common method of visualising volumetric datasets. The most popular algorithm for the construction of the isosurface polygons is Lorensen and Clines [29] marching cubes algorithm. This algorithm uses a hierarchical tree of boxes which intersect the specified density fields in the volume with increasing accuracy. Once a high

degree of intersection accuracy has been maintained each box is converted into one or more surface polygons.

Numerous improvements have been made to this initial algorithm [11,37]. Wilhelms and Van Gelder's method [11] is notable for its use of octrees. Much of the theory and advantages of their method carry over to our new algorithm presented in this dissertation.

There are however numerous problems with isosurface methods (a comparison can be found in [38]):

- Often the data size of the polygons exceeds the size of the entire volume.
- Only discrete density levels may be visualised and interior and amorphous phenomena cannot be visualised.
- Slicing of the volume is generally not naturally supported and CSG-type operations with the volume data are computationally complex.

Thus, although this method has historically enjoyed a lot of support it is now being replaced by more flexible (and recently, faster) methods.

2.5.5 Frequency domain techniques

The section on volume representation methods mentioned the use of frequency domain storage of the volume. This is made possible by exploiting the *Fourier Projection Slice* theorem.

This states that a projection of a volume may be obtained by first converting that volume to a three dimensional frequency space representation, and then extracting a slice from it (the orientation depends on the desired viewing parameters). Once this two dimensional slice is inverse-transformed back to the spatial domain an image is generated which is a projection of the volume. This image does not however contain shading or conventional occlusion characteristics.

The main impetus behind using this method is to achieve faster rendering times as the extraction of a slice of data is a lot faster than a full spatial domain rendering. This is even more true if the volume is continually maintained in its frequency space representation.

Totsuka and Levoy [39] presented a method for achieving better rendering results by performing shading calculations within the frequency space volume.

This method has not enjoyed much success due to the poor quality of the images generated and the large computation times involved.

2.5.6 Direct volume rendering

In recent years a lot of attention has been given to direct volume rendering. As mentioned in the *Background* subsection, this consists of directly rendering the discrete volume array onto the image plane.

The direct volume rendering methods generally fall into one of the following categories:

- *image-order algorithms* - the image pixels are traversed and the corresponding voxels are located and projected normally in a front-to-back order. An example of this is ray tracing.
- *object-order algorithms* - the volume object is traversed and voxels are transferred onto the image plane, often in a back-to-front order. An example of this is the projection method.

2.5.6.1 Ray Tracing

A reduced form of ray tracing called *ray casting* is used for the rendering of volumes. Using this method no secondary reflection or refraction rays are calculated, only the primary ray is used [31].

As with conventional ray tracing, a ray is cast from the image plane into the scene (world coordinates). These rays are generally transformed into object coordinates and some form of differential analyser is used to scan convert the ray through the volume.

The STICKS algorithm proposed by Montani and Scopigno [22], compresses the volume into run-length encoded *sticks* along the z-axis of the volume. The encoded volume was then ray-traced. However due to the single axis compression technique, scan conversion of a ray resulted in a very high number of accesses to their data structure. Reported rendering times were in the order of 250 seconds. More recent approaches to run-length encoding [24] have suggested maintaining three copies of the volume, each compressed along a different axis. This however obviates the advantages of the superior compression achieved through run-length encoding.

Yagel and Kaufman [40], presented their template based volume ray tracing algorithm, which achieved a large reduction in rendering times (rendering a standard medical volume took around 30 seconds). However their algorithm did not use a compressed data structure, so the full volume array had to be maintained in memory.

Sobierajski and Kaufman [41] also presented a very photo-realistic ray tracing algorithm which makes intelligent use of bounding boxes and distance sorting. Rendering times in the order of 6 minutes were reported.

A very effective volume compression and ray tracing algorithm was presented by Ning and Hesselink [27]. As mentioned in the subsection on *lossy compression*, above, their algorithm used vector quantization to produce very high compression ratios. They reported rendering times of 54 seconds for a standard medical dataset. Unfortunately their algorithm did result in very noticeable artifacts in the rendered image, brought about by the compression.

Other methods for accelerating ray tracing of volumes include: Levoy's [13] octree encoding scheme, Zuiderveld's [35] Ray Acceleration by Distance Coding, Avila's [42] near and far ray boundaries (computed by orthogonal z-buffers), and Yagel's [43] coordinate buffer for accelerating ray casting during animation of a volume.

Normally the ray tracing operations mentioned above are performed using parallel rays (resulting in an orthographic projection). The main reason for this is the performance improvements gained by each ray being cast in exactly the same direction. Other reasons include the lack of object scaling with

distance allowing direct measurements to be made of the model. However Novins, Sillion, and Greenberg [44] motivate for the use of perspective projection ray tracing. They presented an adaptive *ray-splitting* process which accelerates the perspective ray casting process and makes it more accurate. Rendering times of 356 seconds for a fairly small medical data set were reported.

2.5.6.2 Projection

A more recent approach to direct volume rendering is the projection or *splatting* approaches. With these methods areas of the volume are directly projected (or *splatted*) onto the image plane, in such a way that occlusion is maintained.

Wilhelms and Van Gelder [45] presented one of the earlier approaches to projection, with their coherent cell projection method. Each cell (a region bounded by eight neighbouring voxels) is treated as a small cube which is parallel projected onto the image plane. The polygons of pixels representing the projection of the cube are approximated using various interpolation techniques ranging from Gourard shading to exponential linear interpolation. These approximated polygons later became known as *splats*. The polygons were projected onto the image plane using either a back-to-front compositing operator or a front-to-back compositing operator. Details of the *over* compositing operator can be found in Porter and Duff's paper [46]. Rendering times of 830 seconds were presented when using hardware Gourard shading and a fairly small volume.

Laur and Hanrahan [12] used the splatting approach to develop a progressive refinement rendering algorithm. This algorithm represented the volume using an octree structure. Hierarchical renderings of the object were obtained by splatting entire regions of the volume as single interpolated polygons. Their octree data structure stored the errors resulting from approximating certain regions by single polygons, so given an overall *acceptable* error the algorithm will traverse the octree to a set depth and splat the corresponding polygons. The techniques presented in their paper were a major motivation for the incremental rendering algorithm presented in this dissertation. Moderate quality images were achieved in only 5 seconds using Gourard shading hardware.

Recently the Shear-Warp projection algorithm was presented by Lacroute and Levoy [24,25]. This algorithm performs a run-length encoding compression of the volume, but uses a simultaneous object-order and image-order traversal to rapidly splat entire slices of the volume onto the image plane. Rendering times of 1 second were reported for a standard medical volume. Unfortunately the algorithm suffers from the same memory problems that the original STICKS ray tracing method suffered from, due to the difficulty in choosing a suitable axis along which to perform the compression. The basic technique of their algorithm is used in the efficient octree algorithm presented in this dissertation, however the memory limitations are removed through the use of octrees. Also due to the use of octrees other possibilities such as incremental rendering become possible.

2.5.6.3 Texture Mapping

Through the use of very high-end graphics workstations which support real-time texture mapping, a very efficient direct volume rendering system can be developed.

Cabral, Cam, and Foran [47] presented an algorithm using inverse Radon transforms to dynamically construct texture maps Using a Silicon Graphics Onyx Graphics Supercomputer rendering times of 0.1 seconds were achieved. The main problem with their method is of course the reliance on very high-end rendering hardware. Their work derives from some original work by Cullip and Neumann [48] in using the Silicon Graphics *Reality Engine* for a texture mapped approach to volume rendering.

Lippert and Gross [49] also use a texture mapping approach for a wavelet based volume rendering algorithm. Their algorithm unifies the use of the frequency space approach and the wavelet decomposition approach. Unfortunately the images produced are still fairly low quality.

2.6 Shear-Warp Algorithm

The Shear-Warp Factorisation algorithm was presented by Lacroute and Levoy [24,25] as a highly efficient volume rendering solution. Their algorithm achieves rendering rates of 1 second per frame for a standard sized medical dataset. Volume representation is through a run-length encoding method which achieves reasonably high compression ratios without losing any necessary information. (i.e. The run-length encoding compresses runs of unused voxels.)

Their algorithm actually consists of a number of other improvements to the whole process of preparing and rendering the data.

2.6.1 Min-Max Octrees for Classification

Often during the manipulation of a volume, the data needs to be re-classified (because isosurface levels have changed, say) and this can be a time consuming exercise. The authors presented a partial solution to this problem in the use of Min-Max octrees to accelerate the classification process.

Essentially their algorithm constructs an octree which covers the entire volume. Each node of the octree contains the bounds of the parameters to the opacity transfer function. Thus, during traversal of this octree, entire regions of the volume which will be transparent can be skipped from classification by integrating the resultant opacity between the bounds of the parameters for the region. They perform this integration efficiently using pre-computed summed-area tables.

2.6.2 Parallel and Perspective Rendering

Both parallel and perspective rendering of the volume is supported by the Shear-Warp Factorisation algorithm. The focal point of the algorithm is the factorisation of the viewing transformation matrix into two much simpler matrices depending on whether the projection is parallel or perspective. The factorisation is as follows:

- Parallel transformations may be factorised into a three dimensional shear matrix which shears the slices of the volume in the X and Y directions only; and an affine two dimensional warp matrix.
- Perspective transformations may be factorised into a three dimensional shear and scale matrix which shears and scales the slices of the volume in the X and Y directions; and a perspective two

dimensional warp matrix. (The scaling of the slices are such that they get smaller as their Z-coordinate increases)

The result of this factorisation is that the rendering of a volume can be done in the following stages:

1. The view transformation matrix is factorised and the scaling and shearing coefficients are calculated as well as the warp matrix.
2. An intermediate compositing image is created, where the slices are composited in Z-order. Each slice is scaled and sheared accordingly before it is composited.
3. Once all slices are composited, this intermediate image is warped into the final image.

Thus the rendering of the volume has been split up into two highly efficient operations: (1) a simple parallel splatting of slices onto the intermediate image, and (2) a rapid two dimensional image warp.

Another advantage of this approach is that the scanlines of the volume are constantly aligned with the scanlines of the intermediate image. This allows for simultaneous image-order and object-order traversing, thus gaining the advantage of both. The image order traversal allows the algorithm to skip rendering voxels when corresponding image pixels are already opaque (front-to-back compositing is used). The object order traversal allows the algorithm to skip image pixels when the corresponding voxels are transparent.

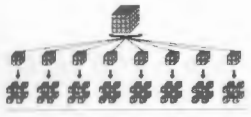
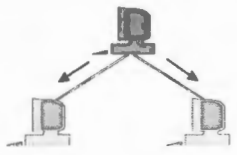
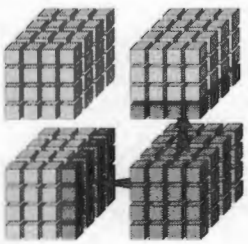
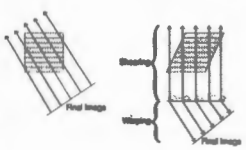
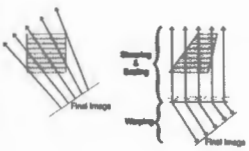
The run-length encoding storage of the volume was chosen specifically to exploit this simultaneous image-object scanline rendering. Unfortunately due to the non-symmetric nature of the run-length encoding (it is a one dimensional encoding along a single axis), the algorithm breaks down when the volume is rotated and the scanlines of voxels are no longer compressed in a direction parallel to the intermediate image. To alleviate this problem the authors suggested maintaining three transposed copies of the volume, with each one being run-length encoded. This makes the runtime memory usage of the algorithm very high.

The authors also presented methods for depth cueing and simple shadow generation during the rendering process. References to the methods of affine and perspective image warping, used during the warping stage, can be found in [50,51,52].

The rendering algorithms presented in this dissertation are based on the Shear-Warp algorithm, but make use of hierarchical data structures instead of the run-length encoding scheme previously used. In doing so, more than a two fold memory decrease in memory usage is achieved without compromising the efficiency of the rendering process.

2.7 Algorithm Overview

Before we proceed with an in-depth technical discussion of the techniques used in our algorithm an overview of the entire algorithm is presented which is intended to serve as a map to guide the reader through the following chapters. The algorithm can be broadly divided into five stages:

	The raw data set is parsed, filtered, and then compressed using an octree based compression scheme eliminating the storage of unnecessary voxels. The octree is designed to contain information in its non-leaf nodes to permit approximation of regions of data easily. In a networked environment this process would occur on a server.	§3.2 and §3.3
	The compressed data may now be *streamed* over a network link such that the data may be incrementally read and used at the client side. The following stages all work with incremental volume data arriving at a client workstation.	§3.3.5
	The compressed data is prepared for rendering (i.e. classified) using a hierarchical classification algorithm which incrementally classifies the data as it arrives. (This process involves computing voxel opacity, colour, spatial gradients, and gradient magnitudes for each visible voxel in the volume.) The same octree data structure is used to contain the classified data. A specialised cache is used to accelerate the classification process.	§3.4
	The classified data (or what there currently is of it) is rendered using a parallel projection version of the Shear-Warp algorithm which uses octree data as opposed to the original RLE data. Missing data (data which has not yet arrived from the server) is approximated using a trilinear interpolation process. The projected parallel image is then warped using an affine image warp.	§4.2 and §4.4
	The classified data (or what there currently is of it) is rendered using a perspective projection version of the Shear-Warp algorithm which uses octree data as opposed to the original RLE data. Missing data is approximated using a trilinear interpolation process. The projected perspective image is then warped using an perspective (non-linear) image warp.	§4.2 and §4.4

2.8 Conclusion

We have seen that there is a tendency to move visualisation operations onto wide area networks such as the Internet. This is primarily due to the possibilities of collaborative diagnosis by remotely located experts, and also the academic advantages of making multimedia resources available to students and research staff. With the acceptance of the Internet this is most likely to occur through the WWW interface.

A lot of improvement on rendering times of medical volume data has been made over the last 20 years, up to the point that volumes may be rendered in almost real-time on average workstations. This makes the publishing of medical volumetric datasets on the WWW a viable proposition. However a number of problems still remain to be solved.

When a user is dynamically manipulating a volume updates should occur at a consistent rate. This would not be guaranteed if a central server were to perform all the computations and then transmit an image over the Internet as the load on the server is not predictable nor is the transmission delays in sending the images. Thus the rendering of volume images at a central server is not a viable proposition due to possible break down under load and also the delays of image transmission over the Internet. (Thus the rendering should be performed on a user's workstation. However factors such as volume transmission times and runtime memory usage have to be reduced.

Volume transmission time and runtime memory overhead may be reduced using various compression techniques but these techniques impact the rendering performance and quality of the images produced. If rendering of medical datasets is to occur on general workstations, then an efficient rendering method for compressed volumes has to be used, which generates good quality images.

The algorithms which will be presented in the following chapters address these problems by providing a highly efficient hierarchical volume compression scheme which allows the volumes to the transmitted efficiently and to be rendered incrementally as they arrive. The incremental rendering guarantees a shorter user-feedback time which is a desirable feature in any visualisation system.

Chapter 3

Octree Compression and Data Classification

3.1 Introduction

The primary goal of this dissertation is to produce an efficient volume rendering system for a client-server model where the bandwidth of the connection may be quite low. The efficiency of data storage and processing time on the side of the server is not an issue as the initial data construction process simply has to be performed once and the results stored. However it is essential that the data structures are as compact as possible and allow for the most efficient rendering once they arrive at the client. The main problem of the construction process is thus the development of this data structure. Our research led us to consider the octree data structure for compressing the volume data, which we found gave significant improvements over some other data structures (such as run-length encoding) due to its symmetric nature.

This chapter covers the theory and algorithms for initially constructing the octree data structure, transmitting it to a workstation incrementally, and then classifying the data either incrementally or at a later stage. We introduce the octree data structure (§ 3.1.1) and develop it for our own purposes in § 3.2. The octree data structure is transmitted incrementally to a workstation (§ 3.3), where the data is classified (§ 3.4). This data structure will carry through to the next chapter where it is used to render or approximate the volume data in an efficient manner.

3.1.1 Octrees for Representing Volumes

One of the main conjectures of this thesis is that the use of an octree data structure for representing the volume is an efficient lossless scheme, in terms of both memory usage and complexity. The octree is a hierarchical data structure which can be viewed as the recursive sectioning of a cubical region of three space into eight smaller cubes. This sectioning is achieved by sectioning the original cube through the centre along each of the three axes. The octree can however also be viewed as a standard tree, where there are eight children to each node in the tree. Both the representations are depicted in Figure 3.1.

Each node thus represents a sub-volume. (The terms node and sub-volume will be used interchangeably throughout this text, for referencing both the data-structure as well as the 3-dimensional sub-volume, or cube.) It is useful to keep both these perceptions in mind as the algorithm develops.

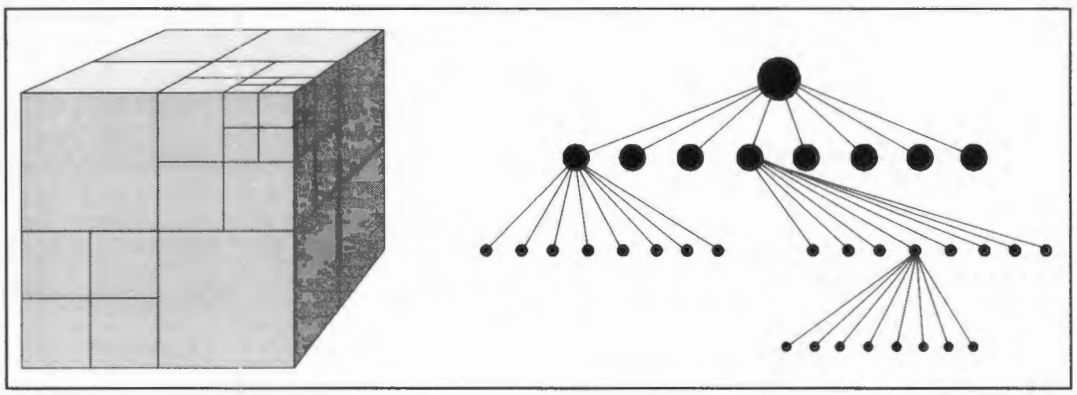


Figure 3.1 - Two depictions of the octree data structure.

An extensive and useful reference for understanding octrees and other spatial data structures is Samet's book [53]. In his terminology the octree which our algorithm constructs is a *bucket PR octree* or more concisely a *region octree*. He makes use of CSG examples to illustrate the use of octrees for representation of both surface based or voxel based data.

The nature of the octree is such that it closely (depending on the depth of the octree) represents structure in the volume. The octree algorithm presented here builds an octree which covers the "useful" areas of the volume and then only stores the data for those regions.

The nodes of the octree also contain information about the nature of the nodes below it or the data inside the node. This allows for approximate rendering of missing nodes later on in the algorithm.

Another advantage of this octree data structure is that the actual octree data structure can be maintained separately from the raw data (unlike RLE compressed volumes). This allows duplicate octree structures to be used (with different node information) during the various stages of volume visualisation.

3.1.2 Classification

Before the raw data is rendered it needs to go through a stage where the spatial gradients and normal vectors at each voxel need to be calculated. This stage is known as classification and can also involve further reduction of the data when the resultant opacity of voxels becomes very small.

As mentioned in the previous section, the data is transmitted to the workstation incrementally. In order to render this data as it arrives it also has to be classified incrementally. This poses some problems as

the calculation of spatial gradients and normal vector requires the referencing of voxel neighbours. Due to the fact that the volume is arriving incrementally it is possible that a particular voxels neighbour is not present and thus only an approximation of its spatial gradient can be made. Then later on, when the neighbour of that voxel arrives, a more accurate spatial gradient can be calculated.

In general it was found that the classification of volume data represented by an octree is a lot slower than classification of data compressed using an RLE scheme. This is primarily due to the difficulty of locating neighbours of voxels under certain circumstances. However due to the incremental nature of the classification algorithm the duration of initial classification may be amortised into the duration of transmission.

3.2 Data Structure

The data structure which contains the information for transmitting and rendering the volume data is split into the three main sections depicted in Figure 3.2. The order of these sections is such that the most fundamental data is at the front of the structure. This facilitates incremental transmission.

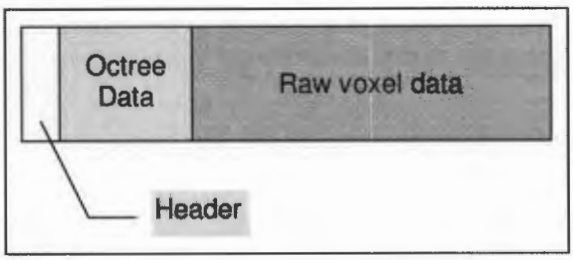


Figure 3.2 - Full data structure of an hierarchical volume.

This data structure needs to contain all information concerning the rendering of the volume and traversal of the octree data structure. Due to the fact that it is incrementally transmitted, the header section (which contains essential start-up information for both octree traversal and rendering) is first, followed by the octree data structure (used for approximating the volume in the absence of raw data and also for making normal rendering more efficient). The last data in the structure is the raw volume data corresponding to the leaf nodes of the octree. This data can only be rendered once the corresponding octree node has arrived (hence it comes after the octree structure), and the data for each node is placed in the same order as the octree nodes themselves.

The first section (a header structure) contains the following information:

Stage	The format of the raw volume data, which should be either RAW or CLASSIFIED. When the data is transmitted to the workstation it should be in RAW.

LowCutoff	The low range cut-off value. (See section 3.3.2)
HighCutoff	The high range cut-off value. (See section 3.3.2)
MaximumValue	The maximum value found in the entire volume. This allows the rendering algorithm to rapidly decide not render a volume if it is going to be completely transparent.
MinimumValue	The minimum value found in the entire volume.
MaximumGradient	The maximum gradient in the entire volume. This is also used for checking for complete transparency.
MinimumGradient	The minimum gradient found in the entire volume.
OctreeDepth	The depth of the octree.
XDimension	The X dimension of the original volume in number of voxels.
Ydimension	The Y dimension of the original volume in number of voxels.
ZDimension	The Z dimension of the original volume in number of voxels.
OctreeDimension	The dimension of the sub-volume represented by the root node of the octree. This sub-volume will always be cubic, and the dimension will always be greater-than or equal-to the any of the other dimensions as well as being a power of 2. (This allows for efficient computation of co-ordinates inside the octree.)
OctreeSize	The number of nodes in the octree.
RawSize	The number of bytes of raw data following the octree.
OpaqueVoxels	The number of opaque voxels in the entire volume.
RootNodeReference	A reference to the root node of the octree. This should always be node 1, but this is used as a sanity check.
NodeCompression	A flag indicating that leaf-node compression has been used on the raw data.

The second section contains the hierarchical octree data structure through which the volumetric data is accessed. In building a data structure for representing an octree, there are two common approaches which are possible:

1. Represent each node in the tree by a data structure which has eight references to child nodes inside it. (This is the classic representation of any data structure which is an instance of a directed acyclic graph.)
2. Represent a node by a data structure which consists of eight nodes information. Each node's information then references another instance of this data structure.

The octree algorithm presented here chooses to use the second representation for the following reasons:

- The depth of the tree is at least one (it is pointless to not perform any subdivision of the volume at all), so there is no need for the explicit storage of a root node.
- For three directions, it is very simple to calculate the neighbouring node of another node. (It will always be within the same node-structure in these cases.)
- The process of locating a node in the tree will require one less de-reference of node child references.

This data structure is depicted in Figure 3.3.

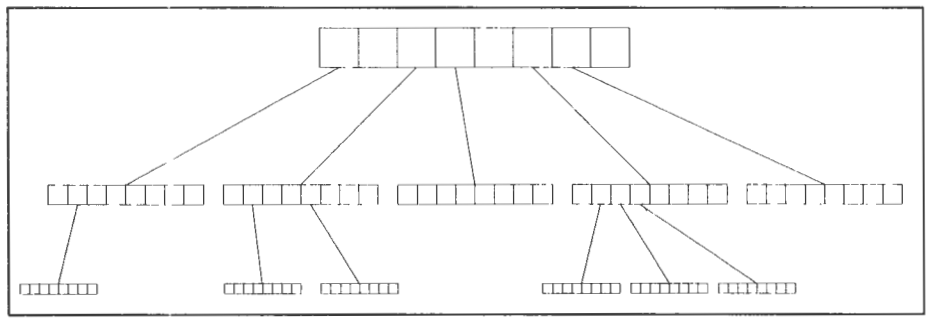


Figure 3.3 - Octree node layout.

The elements of each node structure are as follows:

Child	Reference to a data structure of this format which represents the child nodes of this node. This reference can be: • an actual reference to a node structure, • the value VFULL which represents a leaf-node containing raw data, or • the value VEMPTY which represents a leaf-node containing no data.
DataRef	A reference to the beginning to the raw data for this particular node. This member is only valid if the *Child* member has the value VFULL.
MaximumValue	The maximum value which occurs in any of the nodes below this one, or the maximum value in the raw data of a leaf-node.
MinimumValue	The minimum value which occurs in any of the nodes below this one, or the minimum value in the raw data of a leaf-node.

MaximumGradient	The maximum gradient which occurs in any of the nodes below this one, or the maximum gradient in the raw data of a leaf-node.
MinimumGradient	The minimum gradient which occurs in any of the nodes below this one, or the minimum gradient in the raw data of a leaf-node.
GradientCalculated[6]	Flags for each of the slabs of voxels which cover each of the faces of this sub-volume. The flag can be: • **TRUE** - If all the voxels on this particular face have been correctly classified. • **FALSE** - If some or all of the voxels on this face have not been correctly classified.
GradientApproximated[6]	Flags for each of the slabs of voxels which cover each of the faces of this sub-volume. The flag can be: • **TRUE** - If all the voxels on this particular face have been classified approximately. • **FALSE** - If some or all of the voxels on this face have not been classified at all.
AverageValue	The average value of all voxels in all nodes below this one.
AverageGradient	The average gradient of all non-transparent voxels in all nodes below this one. (This is only calculated during classification as it may alter due to different opacity settings.)
AverageNormal	The average normal vector for all non-transparent voxels in all nodes below this one. (This is only calculated classification as it may alter due to different opacity settings.)
IsCompressed	If this node is a leaf-node then this flag signifies whether the raw data contained in this sub-volume is compressed further or not. (This is useful as sometimes run-length compression can result in the compressed volume being larger than the original, and so in this case it is simply stored as-is.)
CornerValues[8]	The values of the voxels at each corner of this sub-volume. These values are used during rendering to perform trilinear interpolation over a sub-volume, when the raw data is not available. The calculation and storage of these values only occurs during the classification stage as this data is not required before.

The final full set of data structures which represent the octree, simply form an array of these node structures. Each of the child references in the nodes are simply indices into this array. This approach is used (rather than direct pointers) as it allows easy duplication of the octree structures for other purposes. In the next chapter, a duplicate octree structure is used to represent the modified structure due to classification, as well as to store information pertaining only to the rendering stage.

Lastly, the third section of the data structure contains the raw data which is referenced by the octree. Some of the data may be run-length compressed.

3.3 Octree Construction and Incremental Transmission

3.3.1 Filtering Phase

The first stage of building the compressed volume data structure, involves the filtering of the original raw volumetric data. Generally, most data generated by medical scanners, has 16-bit voxels. In other words each voxel may have an integer value in the range -32767 to 32768. However this entire range of values is seldom used, and the "sub-range" varies from scanner to scanner and the scanning methods used.

For the sake of memory efficiency the entire algorithm presented here operates on 8-bit voxels, where the density value ranges from 0 to 255. It is thus necessary to convert this original data into an 8-bit form and also to optimise the range of values to better cover the "interesting" data. This has a slight drawback in that if the original data used a very wide range of values then some information might be obscured in renderings of the 8-bit volume. However the efficiency advantages from using 8-bit values are great, and the dynamic range of the rendered images was found to be sufficient to display most phenomena occurring in standard medical volume datasets we tested. All the algorithms in this dissertation could however be easily extended to more than 8-bit representations.

The method by which this is achieved in similar to a process which is often used in image processing, called a contrast-stretch. In image processing an image with low contrast, has a lot of its information content gathered together in a very small band of values. Once the range of this band of values is known a contrast stretch algorithm can be used to spread the values over the entire range which the image supports, thus giving the image better contrast.

The filtering algorithm for volumes takes as input the following parameters:

- The size of an individual voxel in bits.
- A flag denoting signed voxels.
- The lower limit of the useful values in the volume. (Called *LowerLimit* below.)
- The upper limit of the useful values in the volume. (Called *UpperLimit* below.)
- A voxel mask, which is applied before the comparisons with the limits are made.

The choice of the *LowerLimit* and *UpperLimit* values are made through analysis of a histogram of the data inside the volume. Most of the histogram should appear flat, and only in a small range should there be peaks. These values are optimally placed on either side of these peaks. This is depicted in Figure 3.4.

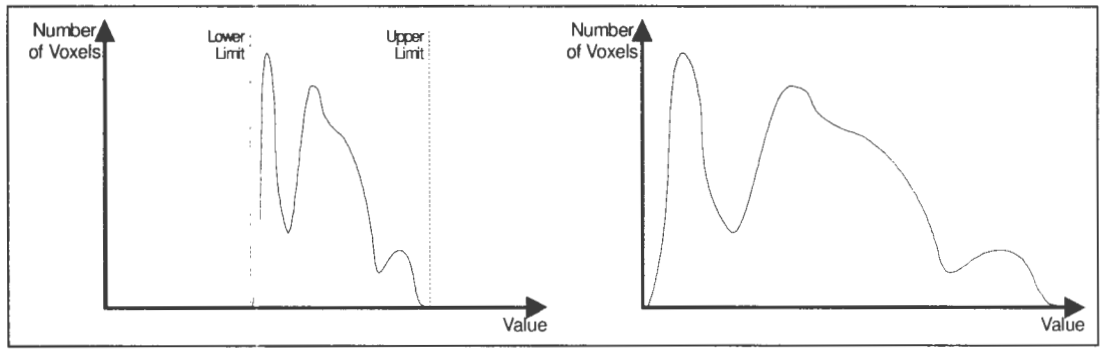


Figure 3.4 - Comparison of histograms before and after contrast stretching.

The contrast stretch algorithm is then applied throughout the volume to generate a volume whose values lie in the range 0 to 255, and which covers exactly the range of useful values.

To perform this operation rapidly, the following algorithm is used:

1. Create an array with an index that covers the full range of values in the original volume.
2. Fill the array with the value 0 for all indices less than *LowerLimit*.
3. Fill the array with the value 255 for all indices greater than *UpperLimit*.
4. Fill the array with the value V for all indices between *LowerLimit* and *UpperLimit*. The value of V is given by the formula, $V = \frac{255 - (CurrentIndex - LowerLimit)}{UpperLimit - LowerLimit}$.
5. For every value in the original array replace its value by the value in the array given by using it as an index.

It should be noted that the lower and upper limits used here are not the same as the ones used for determining the isosurface. The isosurface values serve to further refine the range specified by these values and are thus in the [0, 255] range.

3.3.2 Construction Phase

The next stage of building the compressed volume is to construct the octree around the filtered data from the previous step. This stage requires three parameters:

- A maximum octree depth. (See chapter 5 for determination of the optimum depth.) It is worth noting that considering this is an octree the size of the tree increases with $O(8^N)$.
- A minimum threshold value which specifies the lower limit of the range of values which we wish to extract from the volume and render.

- A maximum threshold value which specifies the upper limit of this range of values.

The algorithm then proceeds recursively to build the octree as follows:

1. Recursively subdivide the volume into eight sub-volumes (see Figure 3.1).
2. Once the recursion is equal to maximum tree depth, stop recursing.
3. Check if any of the values in the sub-volume are in between the two threshold limits. If they are then return the value VFULL and a reference to the data, back to the previous level of recursion.
4. If there are no values in the range then return the value VEMPTY.
5. After the result of each recursive call to the level below check the results of each call. (There should be eight in total).
6. If each one returned VFULL then simply return VFULL.
7. If each one returned VEMPTY then simply return VEMPTY.
8. If some calls returned VEMPTY and others not, then create an octree node, where each of the node items contains the value VEMPTY of VFULL (depending on whether the call for that region returned VEMPTY of VFULL) as well as the reference to the data that was returned. Then return the reference.

Once this recursive process is completed an entire octree structure will exist, which encompasses the volume according to the threshold limits specified. However this octree data structure is such that the nodes are stored in "reverse order". (i.e. The nodes at the beginning of the node array represent sub-volumes at the bottom of the tree, and the root node lies at the end of the array.) This will not work for incremental transmission, as the most general nodes need to be transmitted first. Thus a reordering of the octree nodes is required.

Not included in the above algorithm, are the determinations of maximum and minimum values for each octree node. These values are simply passed back up the recursive tree in a similar fashion to the data references.

3.3.3 Reordering Phase

This phase consists of a breadth-first traversal of the octree, where the nodes traversed are placed into a new array in order of traversal. This order is necessary as this allows the algorithm on the workstation to incrementally refine a rendering of the volume as it arrives.

Initially only the top sub-volume will exist forcing an approximation of the entire volume by a single interpolated cube. Then when another level arrives that cube may be broken into eight smaller cubes which will better approximate the volume. Then again, each of these eight cubes will be broken down further into eight cubes each, and so on.

3.3.4 Node Compression

At this stage the first two main sections of the entire data structure (see Figure 3.2) have been created. It is now necessary to create the raw data section, by selecting only the data referenced by the octree, from the original volume. It was found experimentally that, especially for lower octree depths, the data compression brought about by the octree is far from optimal (See the *Experimental Results* chapter for an analysis of this). Further compression (in the form of run length coding) is performed on this raw data before it is placed into the main data structure.

The algorithm at this stage proceeds as follows:

1. Recurse down the octree until nodes with child pointers of value VFULL are located.
2. By calculation of the position of the sub-volume represented by this node, locate the corresponding data position in the original three dimensional array of raw volume data.
3. Compress the raw data only in the region identified by the size of this sub-volume, using a run length encoding of each X-axis aligned stick of voxels.
4. If this compression results in a data set larger than the original then stop compressing and simply store the data directly into the main data structure at the next available position.
5. If the compression is successful then store the compressed data at the next available position in the main data structure.
6. Store a reference to this data into the child node structure, and continue the recursion.

Once completed, the main data structure will now have all three sections complete and the data is ready for transmission.

It should be noted that this node compression only exists while the volume is being transmitted, and as soon as the data arrives at the workstation the data for each leaf node is decompressed. The reason for this, is that the advantage of using the octree data structure over a full RLE compression of the volume data is the symmetry of the data. However if some nodes are stored using an RLE compression in the X-axis then this symmetry is broken and the advantages of this algorithm are lost.

3.3.5 Transmission

The server may now transmit the main data structure byte-for-byte to the workstation. The incremental renderer on the workstation may then start rendering the data as soon as the header and the root node of the octree arrive. (This constitutes about 204 bytes of data.) The workstation should then render repeatedly, as often as possible, the data as it arrives.

The classification and rendering algorithms presented in the following sections, check whenever they reference an octree node or the data referenced by a leaf, to see if that data actually exists. If it doesn't then they take steps to approximate that data.

3.4 Incremental Classification

The introduction mentioned the problem that is experienced during classification, caused by the difficulty in locating neighbouring voxels in an efficient way. Generally this classification process should only be performed once, as the data arrives at the workstation, however a user of the system is free to change the classification settings at any stage, which would require a reclassification. The original Shear-Warp algorithm using the RLE compressed volume, did not have this neighbour locating problem, however due to the asymmetry of the compression, three transposed copies of the classified volume have to be generated every time the volume is classified. This process combined with the original classification process could end up taking as long as the octree classification algorithm presented below.

Another advantage of the octree data structure which has not yet been mentioned is that with the RLE compressed volume, the original volume (pre-classification) has to be maintained separately if the classification process is to further compress the volume, allowing for rapid skipping of empty runs. However with the octree algorithm, a duplicate octree structure can be used, after classification, which has been modified to rendering of large transparent areas. This is yet another reason why the octree representation is more space-efficient than the RLE representation.

3.4.1 Caching Mechanism

The problem with octree classification arises when a voxel on the edge of a particular sub-volume needs to be classified. In order to classify the voxel its spatial gradient needs to be calculated. A simple first order spatial gradient is used, given by,

$$\nabla(x,y,z) = \begin{pmatrix} V(x+1) - V(x-1) \\ V(y+1) - V(y-1) \\ V(z+1) - V(z-1) \end{pmatrix}$$

where V is a discrete function returning the value of a voxel at the (x,y,z) location in the volume. Thus in order for this formula to be evaluated the six neighbours of the voxel have to be located. If the voxel at (x,y,z) lies at the edge of a particular sub-volume then up to three neighbouring sub-volumes have to be referenced to locate neighbours.

One approach to making this more efficient would be to directionally thread the octree. This involves each node containing six references (one for each face of the sub-volume), to each of the neighbouring nodes. The problem with this method is that some nodes may have many smaller nodes as neighbours at certain faces. The only solution in this case is to store a reference to one of the parent nodes of these smaller nodes, which is at the same level of the octree as the node in consideration. Thus octree traversal is still necessary, and at the expense of adding six additional references to each node.

The solution which is proposed in the octree algorithm is to maintain a cache of recently referenced nodes, and to provide an efficient method of locating these cached nodes depending on the co-ordinate of a voxel. The main argument behind using this method, is that during classification the voxels on

each face of the sub-volume are classified separately and in order of face. Thus only a few nodes are neighbouring a particular face they will be constantly accessed until that entire face is classified. It is thus advisable to have these nodes in a cache. Another potential solution is proposed in §6.3.

The cache is implemented as an LRU (least recently used) cache, with a hashing function H(x,y,z) which codes a co-ordinate for each node in the cache allowing for efficient searching, given only a voxel location. The problem with this hashing function is that it has to maintain order in three-space.

The function H(x,y,z) is implemented in the following way:

1. Shift each x,y,z value so only the lowest 10 bits are used. (This is generally not necessary as most volumes considered are under 1024x1024x1024 in size.)
2. Create a new 30-bit value given by, alternately placing the bits of x, y, and z at each successive bit position. The resulting value will thus appear as in Figure 3.5.
3. Return this value as the hash value.

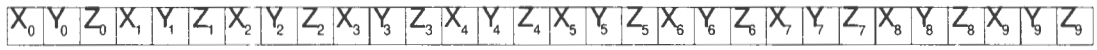


Figure 3.5 - Construction of hash value, shown as a sequence of bits.

This bit encoding of the co-ordinates actually represents sets of eight possible choices (given by each successive set of 3 bits) which can be made at each level of the octree when traversing it. See [54] for a full explanation of this algorithm and for its other applications.

Whenever a node is added to the cache, a hash value in generated using the co-ordinate of its (x-minimum, y-minimum, z-minimum) corner. Then when a search for a particular voxel has to be performed, a hash value for the voxels location is generated, and a binary search is performed on the elements to the cache to locate the closest value (but lower) to the voxels hash value. Once this node is found a check is made using the stored dimension of that node, to see if the voxel lies within it. If it doesn't then the node which contains that voxel does not lie in the cache, and it has to be searched for using an octree traversal.

The *Experimental Results* chapter (Chapter 5) has a section on the determination of the optimum cache size for our algorithm, as there is obviously a trade-off between cache searching time and the traversing of the octree.

3.4.2 Algorithm

In order to classify the volume the user has to supply an opacity transfer function. This is simply a function which specifies the opacity of any voxel in the volume given its value and its gradient. In more advanced volume rendering systems (say using multimodality information) other parameters can also be used. In the algorithm presented here this transfer function is calculated by,

$$Opacity(value, gradient) = ValueTable[value] \times GradientTable[gradient]$$

where the *ValueTable* and *GradientTable* are simply arrays with indices in the range of all possible value and gradient values, and which contain opacities from 0 to 1. The user then only specifies these two tables.

The algorithm for classifying the volume is presented below. It uses the following primary functions:

- *Classify* - Main function called to classify a volume dataset represented by an octree.
- *PerformClassification* - Performs classifications operation for a particular node in the octree.
- *SpatialGradientCached* - Returns the spatial gradient at a particular voxel, using the node cache for efficiency.
- *LocateInCache* - Locates the node containing a particular voxel using the node cache. If it is not in the cache then it is located in the octree and then inserted into the cache.

Details of the algorithm follow...

```
PROC Classify(root_node, value_table, gradient_table)
    InitializeCache();
    new_root = DuplicateOctree(root_node);
    PerformClassification(root_node, new_root);
END
```

The *Classify* function accepts the root of the octree (to classify) as a parameter, as well as the opacity transfer function tables: *ValueTable* and *GradientTable*. (These would normally be passed through to the actual classification function below, but we will omit this for simplicity.)

Firstly the node cache is initialised. The *InitializeCache* function simply clears the node cache and sets all the LRU (least recently used) values to a null value.

Next, the entire octree data structure which was passed to *Classify* is duplicated. This allows the classification process to store different information into this octree, specifically for the rendering process. (Note that this has not duplicated the actual raw data referenced by the octree.)

Lastly the recursive classification of the volume is begun by calling the *PerformClassification* function with the root nodes of the two octrees. This function will then proceed to classify the entire volume by performing a depth-first traversal of the octree.

```
PROC PerformClassification(old_node, new_node, flag_unclassified)
    IF (NOT flag_unclassified) RETURN;
    flag_opaque      = FALSE;
```

The *PerformClassification* function takes two node references as parameters, one from the original octree and one from the duplicated octree. A reference to an *unclassified* flag is also passed. This is to return to the parent node (processed by the calling function) information about whether or not the node and all of it's children have been successfully classified or not. The initial value of this flag (once dereferenced) should be equal to the current setting for this node.

The first step in this function is to check the *unclassified* flag. If it is set (TRUE) then that indicates that this entire branch of the octree has already been classified and no further processing is required. In this case the function returns immediately to the caller.

Next, an *opaque* flag is cleared. This flag indicates whether the entire node and all of its children represent voxels will be completely transparent using the current opacity transfer function. The flag is initially cleared and then if any opaque voxel is encountered it is set.

```
IF (old_node.value = VFULL)
    new_node.corners = ExtractCornerVoxels(old_node.data);
```

If the node which was passed to this function represents a leaf-node of the octree which contains raw voxel data, then this data is now classified. The first step is however to extract the values of the corner voxels from the original raw data referenced by this node. The function *ExtractCornerVoxels* references each of the eight corner voxels of the sub-volume of the node. For each corner voxel (called v_0) it locates the three voxels at each of the neighbouring sub-volume's corners (v_x v_y v_z). The value of the corner voxel (v_c) is then calculated by,

$$v_c = \frac{3v_0 + v_x + v_y + v_z}{6}$$

This process is repeated for each of the eight corner voxels, such that a list of eight v_c values is returned.

```
IF (NOT GetClas(old_node,(1,1,1))

    FOR ((x,y,z) = (1,1,1) TO (old_node.(dim_x-1, dim_y-1, dim_z-1))

        (xdiff,ydiff,zdiff) = SpatialGradientNoCache(old_node,(x,y,z));

        old_node.data[x,y,z].normal   = (xdiff,ydiff,zdiff);
        old_node.data[x,y,z].gradient = sqrt(xdiff² + ydiff² + zdiff²);

        IF (IsOpaque(old_node.data[x,y,z].value, old_node.data[x,y,z].gradient))
            flag_opaque = TRUE;
        END
    END
END
```

The first stage of classification now begins by classifying the inner voxels of the sub-volume represented by node *old_node*. Firstly a check is made to see if the first voxel has its unclassified flag set. (It is safe to check only the *(1,1,1)* voxel as this flag will always be the same for all inner voxels of the sub-volume.) If the flag is set then the process of classification is begun. This process takes place inside a third order loop which runs through all the inner voxels of the sub-volume.

For each voxel a spatial gradient is calculated (using the finite difference method mentioned earlier). Their is no need to use the cache at this stage as all the neighbouring voxels are guaranteed to exist within this same sub-volume. The normal vector for this voxel is then set-up according to the spatial gradient and then the gradient value is set to the magnitude of the spatial gradient. If the resultant opacity of this voxel (calculated by using the *IsOpaque* function) is non-transparent then the *opaque* flag is set.

```
FOR (Each of sub-volume's six faces)
    FOR ((x,y,z) = (All coordinates on current face))

        IF (NOT GetClas(old_node,(x,y,z))

            (xdiff,ydiff,zdiff) = SpatialGradientCached(old_node,(x,y,z));

            old_node.data[x,y,z].normal   = (xdiff,ydiff,zdiff);
            old_node.data[x,y,z].gradient = sqrt(xdiff² + ydiff² + zdiff²);

            IF (IsOpaque(old_node.data[x,y,z].value, old_node.data[x,y,z].gradient))
                flag_opaque = TRUE;
            END

            flag_unclassified = flag_unclassified OR (NOT GetClas(old_node,(x,y,z)));
        END
    END
END
```

The next stage of the classification process then proceeds by classifying the voxels that lie on each of the sub-volumes six faces. These are treated separately as the neighbouring voxels lie in neighbouring nodes. A third order loop is used, where the outer loop runs over the six faces of the sub-volume and the inner two loops run over the coordinates of all the voxels lying on the current face.

For each voxel the current classification state (using *GetClas*) is checked. If the voxel has not got its unclassified flag set then it is ignored, as it has already been correctly classified. Otherwise the spatial gradient is calculated (as before) except that now the node cache is used. (The function *SpatialGradientCached* will be covered in more detail below.) The *opaque* flag is also set if the voxel is non-transparent. Finally the *unclassified* flag is set if this voxel has not been correctly classified.

```
RETURN (flag_opaque, flag_unclassified);
```

At this stage the classification of this leaf-node is complete and the data may be returned to the calling function. The *opaque* and *unclassified* flags are passed back. We now continue the details of this function as though the "`IF (old_node.value = VFULL)`" check was unsuccessful.

```
IF (NOT old_node.value = VEMPTY)

    FOR (index = 0 TO 7)

        flag_opaque = flag_opaque OR
                      PerformClassification(old_node.child[index],
                                            new_node.child[index],
                                            old_node.child[index].flag_unclassified);
    END
```

If the node which was passed to this function is an internal octree node (i.e. a parent node) then the *PerformClassification* function is called recursively to classify the child nodes of this node.

This is performed by looping through each of the child node references and calling the *PerformClassification* function with both the old and new octrees child nodes (corresponding to the current child node index). The value of the child node's *unclassified* flag is also set. (This flag is normally passed as a reference as it is modified and passed back.)

```
    new_node.corners[0] = old_node.child[0].corners[0];
    new_node.corners[1] = old_node.child[1].corners[1];
    new_node.corners[2] = old_node.child[2].corners[2];
    new_node.corners[3] = old_node.child[3].corners[3];
    new_node.corners[4] = old_node.child[4].corners[4];
    new_node.corners[5] = old_node.child[5].corners[5];
    new_node.corners[6] = old_node.child[6].corners[6];
    new_node.corners[7] = old_node.child[7].corners[7];
```

Next, the corner voxels for this node have to be calculated. The actual values for the corner voxels will have already been calculated by the leaf-node classifications, so it is simply a matter of copying the correct corners from each of the child nodes.

```
    flag_unclassified = old_node.child[0].flag_unclassified OR
                        old_node.child[1].flag_unclassified OR
                        old_node.child[2].flag_unclassified OR
                        old_node.child[3].flag_unclassified OR
                        old_node.child[4].flag_unclassified OR
                        old_node.child[5].flag_unclassified OR
                        old_node.child[6].flag_unclassified OR
                        old_node.child[7].flag_unclassified;
    END

    RETURN (flag_opaque,flag_unclassified);
```

Finally the *unclassified* flag is set to the logical OR of the *unclassified* flags of each of the child nodes. Thus the flag will be set if any of the child nodes have unclassified voxels in them. The *opaque* and *unclassified* flags are passed back. We now continue the details of this function as through the "`IF (NOT old_node.value = VEMPTY)`" check was unsuccessful.

```
IF (old_node.value = VEMPTY)
    RETURN (FALSE, FALSE);
END
```

If the node passed to this function represents an empty area of the volume then the *opaque* flag and the *unclassified* flags are both cleared and returned to the calling function immediately.

This now completes the details of the *PerformClassification* function. We now look at some of the more minor functions in detail.

```
PROC SpatialGradientCached(node, (x,y,z))

    xdiff = LocateInCache(node.data[x,y,x].value, x+1, y, z) -
            LocateInCache(node.data[x,y,x].value, x-1, y, z);

    ydiff = LocateInCache(node.data[x,y,x].value, x, y+1, z) -
            LocateInCache(node.data[x,y,x].value, x, y-1, z);

    zdiff = LocateInCache(node.data[x,y,x].value, x, y, z+1) -
            LocateInCache(node.data[x,y,x].value, x, y, z-1);

    RETURN (xdiff, ydiff, zdiff);
END
```

The *SpatialGradientCached* function calculates the spatial gradient at voxel *(x,y,z)* in the specified node. This is performed by querying the node cache for the values of each of the voxel's neighbours, and then applying a first order finite difference method. The final 3-vector containing the spatial gradient is returned.

```
PROC LocateInCache(centre_data,(x,y,z))

    node = SearchCache(x,y,z);

    IF (node = NULL)
        node = SearchOctree(x,y,z);

        IF (node = NULL)
            centre_data.flag_NOT_classified = TRUE;

            RETURN centre_data;
        END

        AddNodeToCache(node);
    END

    RETURN node.data[x,y,z];
END
```

The *LocateInCache* function retrieves the value of a particular voxel (given its absolute coordinate in the volume) using the node cache for efficiency. The function accepts a reference to the centre voxel in the preceding spatial gradient calculation, as well as the coordinate of the neighbouring voxel which is required.

Firstly, the node cache is searched (*SearchCache*) to find a node which contains this voxel. If the returned node is valid then its raw data is referenced and the value of the required voxel is calculated. This value is then returned and the function terminates.

Otherwise if the node returned from the node cache search is not valid then a depth first search (*SearchOctree*) is performed on the octree to locate the node containing the required voxel. If the node is located then it is added into the cache (*AddNodeToCache*) and the voxel value is returned. Otherwise if no valid node is located (because the volume has not been fully transmitted yet, or the voxel lies in an empty node) then the *unclassified* flag for the centre voxel is set, and the function returns the value of the centre voxel as an approximation of it's neighbour.

The functions *SearchCache* and *AddNodeToCache* use the methodology described in §3.4.1.

3.4.3 Summed Area Tables

An optimisation which can be used during the classification process is the use of summed-area tables to omit regions of the volume from classification very quickly. This optimisation can be performed when the classification algorithm knows the maximum and minimum values and gradients in a region of the volume. Knowing these values allows the algorithm to integrate the resulting opacity over that range, and if it the result is totally transparent then that region may be omitted from classification.

In the original Shear-Warp algorithm a min-max octree was constructed for the volume just prior to classification for the purposes of using summed-area tables. However the octree data structure already contains these min-max values and thus no extra octree construction is necessary.

A summed-area table is a two dimensional discrete array of values ***S(u,v)*** such that,

$$S(u,v) = \sum_{i=0}^{u} \sum_{j=0}^{v} S(i,j)$$

Using this table the integral over any region of S can be calculated by the following formula:

$$\sum_{u_{min}}^{u_{max}} \sum_{v_{min}}^{v_{max}} S(u,v) = S(u_{max}, v_{max}) - S(u_{max}, v_{min} - 1) - S(u_{min} - 1, v_{max}) + S(u_{min} - 1, v_{min} - 1)$$

Thus if the values u represent values given by *ValueTable[value]* and the v values represent values given by *GradientTable[gradient]*, then the integral of the opacities of a sub-volume may be calculated very efficiently using only the minimum and maximum values and gradients for that sub-volume. If the integral lies below the minimum opacity level (before transparency is assumed), then that sub-volume may be safely skipped during rendering and does not need to be classified.

3.5 Conclusion

Research led us to consider the octree data structure for compressing volume data during its transmission phase and during its rendering phase. This structure was chosen for its three dimensional and symmetric nature which we conjecture will give us faster access and better compression ratios.

The entire volume data structure contains three sections: a header, the octree, and the raw data. The first two sections generally only comprise about 1% of the entire data structure however they contain most of the structural information for the volume. We will see in the next chapter that this will facilitate approximate rendering of the data when only part of the entire data structure is available.

Construction and transmission of this data structure is fairly simple and involves a number of stages which filter the data (according to user specified parameters), construct the octree from the filtered data, and then correlate the raw data to the octree structure. A technique called node compression can also be used to further increase the compression ratio. Considering that this process would generally be performed on a server and then the data structure would be stored in a file for rapid future access, its performance (in terms of memory and processor speed) is not critical. However the results in (§ 5.5) will show that it could feasibly be performed "on-the-fly" as well.

The classification stage for volume data is often a very costly stage in terms of the memory required for the generated data structures. In the case of the Shear-Warp algorithm's RLE data structure the classification process builds a data structure (typically) 4 times larger than the original, and then this the classification has to be repeated twice more (once for each axis), ultimately generating a data structure 12 times larger than the original! The octree classification builds a structure which is only 4 times larger than the original as (due to its symmetry) it does not need to perform classification for each axis.

Unfortunately, the octree classification process is a lot slower than the RLE based process, so a number of techniques such as caching during processing and incremental classification (so the classification is done during the transmission of the volume thus not impacting final rendering times at all) are used. Future research should try to address this problem more efficiently, perhaps through the use of directional threading in the octree. (See §6.3)

Other classification process enhancements such as min-max octrees and summed area tables (common to the RLE method) fit elegantly into the octree framework.

The next chapters presents the theory and algorithms for rendering the volume data which was generated using the algorithms in this chapter.

Chapter 4

Rendering Hierarchical Volumes

4.1 Introduction

Incremental rendering is performed through the use of some form of hierarchical representation of the volume, where different levels of geometry may be rendered which approximate the volume to an increasingly higher degree. The rendering algorithm presented here makes use of the hierarchical octree data structure to achieve this. This however requires the development of an efficient rendering algorithm for octree based volume data. Given that one of the main objectives of the incremental rendering algorithm is for it to run on average workstations a highly efficient algorithm has to be chosen. Recently the Shear-Warp Factorisation [24,25] algorithm was developed which originally used RLE data structures to achieve rendering rates of approximately 1 second per frame. (A lot of the basic theory of the rendering methods used here is also covered in the Shear-Warp algorithm description [24,25].)

This chapter concentrates on the development of an octree based Shear-Warp Factorisation algorithm. It will be shown that this offers many improvements ranging from incremental rendering ability through to vastly reduced runtime memory usage. The original algorithm stored three transposed copies of the RLE data in memory and constantly had to choose between one (due to the asymmetry of the compression) when the volume was rotated. The octree compression is symmetrical so this is not required.

We begin by looking at the unique problems experienced with converting to an octree data structure such as the problem of octree traversal order as well as the problem of spatial filtering. New abilities which the octree provides will be presented such as partial rendering when the octree is not complete. These problems and features are common to both parallel and perspective volume rendering, however some of the basic rendering theory is different making the parallel and perspective algorithms quite different.

For each of parallel and perspective rendering methods, the detailed mathematical theory of the methods will be given as well as the theory of the octree traversal (particular to either parallel or

perspective rendering). The incremental rendering aspects of the algorithm will also be discussed for each method, following which a pseudo-code algorithm will be presented in detail.

4.2 Converting Shear-Warp from RLE to Octree Data Structures

The method of Shear-Warp Factorisation rendering is based around the fact that a 3-dimensional projection matrix may be factorised in the following ways:

- If the projection matrix is a parallel projection matrix then it may be factored into: (1) a 3-dimensional shearing matrix which shears in two directions only; and (2) a 2-dimensional affine warp matrix.
- If the projection matrix is a perspective projection matrix then it may be factored into: (1) a 3-dimensional shearing and scaling matrix which shears in two directions and also scales orthogonally to the shearing direction; and (2) a 2-dimensional perspective warp matrix.

This process is depicted in Figure 4.1. The mathematics of this factorisation will be presented in the following sections.

This "separation" of the projection matrix allows for the efficient rendering of the volume in both image-order and object-order simultaneously onto an intermediate image, followed by a warp of the intermediate image to obtain the final image. This means that during the rendering process both the image and the volume may be traversed in a complimentary order such that regions of the image and volume may be skipped when either a region of the image is opaque or a region of the volume in transparent.

The conversion of this process from using scan-line order data to octree data implies a number of problems such as:

- How is the octree traversed and the data rendered?
- What happens with the filtering processes used in the original algorithm?
- How is the volume rendered when not all the octree data is present?

Most of these problems are due to the fact that the volume data is no longer stored in scan-line order, so it is not possible to scan the volume and the image in exactly the same order.

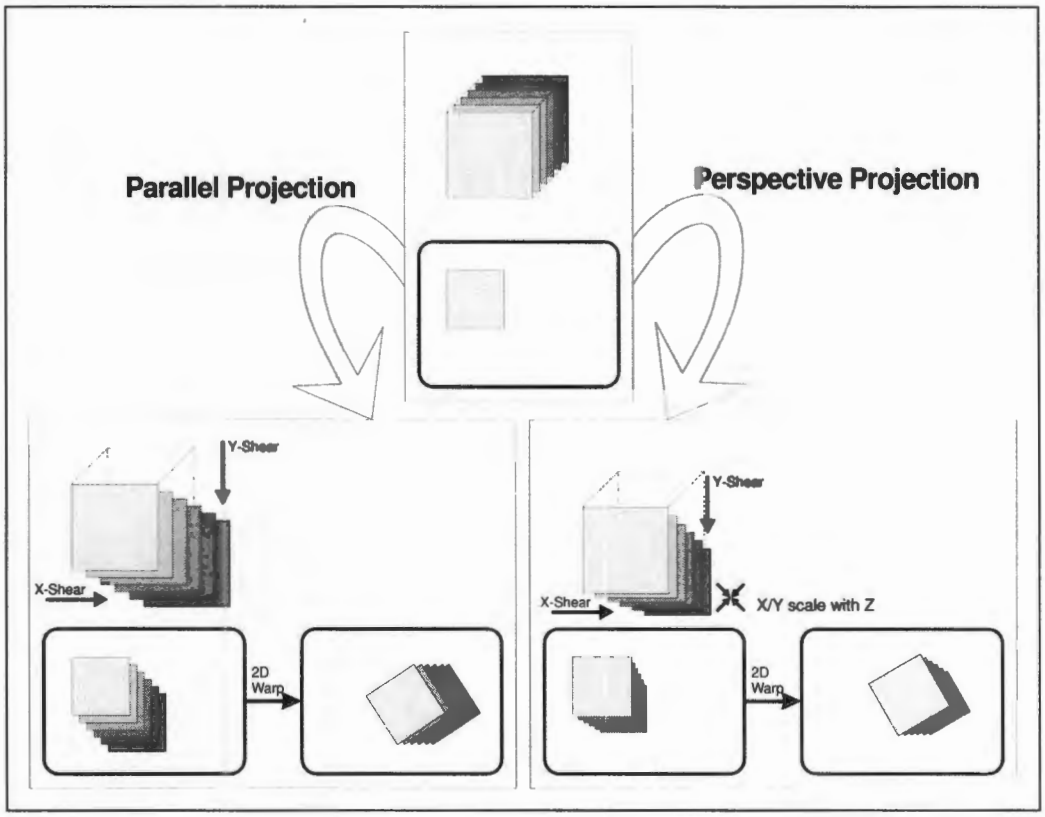


Figure 4.1 - Shear Warp Factorization process for parallel perspective projection.

4.2.1 Traversal Order Problem

On examination of the octree data structure it is apparent that the data contained in each leaf-node of the octree is stored in scan-line order and represents a small cubical region of the volume, so the problem of rendering the entire volume could be perceived as rendering lots of smaller sub-volumes placed in different locations. The solution is not this simple however as certain sub-volumes (represented by leaf-nodes) will be obscuring others, and thus need to be rendered before the ones which they obscure. Also, one of the main advantages of the original Shear-Warp algorithm was the use of a front-to-back compositing buffer, so that voxels near the back of the volume need not be rendered if their contribution to the final image is very low. It is critical to maintain this advantage in our octree algorithm to achieve comparable performance.

In the following sections on parallel and perspective rendering, methods of traversing the octree will be presented which ensure that the sub-volumes are rendered in such an order that occlusion is maintained, and the advantages of the front-to-back compositing buffer are still present.

4.2.2 Filtering

In the original Shear-Warp algorithm bilinear filtering is performed on neighbouring voxels in the volume to achieve an image with reduced aliasing artifacts. Due to the separation of the volume into discrete sub-volumes by the octree, this filtering of neighbours breaks down at the borders of the sub-

volumes. This difficulty comes about due to the need for an octree traversal each time a neighbouring voxel (that lies in a neighbouring sub-volume) is to be located.

There are however both advantages and disadvantages to this filtering process.

As mentioned above, the filtering is bilinear and thus only operates on the current slice of the volume being rendered. The object of this filtering process is to reconstruct the volume integral more accurately however, in order to do this, the filtering has to be performed on all neighbours of a voxel (i.e. trilinear filtering needs to be used). Thus the bilinear filtering is only a partial solution to the problem of achieving an aliasing free image.

The octree algorithm presented here omits this filtering step completely for the sake of efficiency. It was found (see *Experimental Results* chapter) that the omission of this filtering step did not impair the quality of the image by an amount significant enough to justify the extra processing time required for filtering. Future research will attempt to introduce filtering to this octree rendering technique, without impacting the performance dramatically.

4.2.3 Partial Rendering

The primary implication of incremental rendering is that regions of the volume need to be rendered when there is either no voxel information available or only partial voxel information. Due to the hierarchical nature of the octree, it essentially contains multiscale representations of the original volume in order of increasing scale. In other words each level of the octree contains nodes which approximate that region of the volume, and this approximation improves as the depth of the octree increases.

As presented in section 3.2, the octree structure contains information such as the average gradients and values for each of the nodes which can be used for approximating the sub-volumes. However in order to achieve acceptable images it is necessary to store other information as well which allows the sub-volume to be approximated more accurately.

Fortunately, due to the nature of the octree data structure it is completely separated from the volume data (unlike the RLE compressed data). This enables the algorithm to create a duplicate octree structure which is used during the rendering process, and which stores the extra information required. In practice the algorithm uses this second octree to completely remove sub-volumes which will be transparent (due to the current classification settings), without affecting the original data structure in any way, and without making (expensive) duplicates of the entire volume. This process is depicted in Figure 4.2.

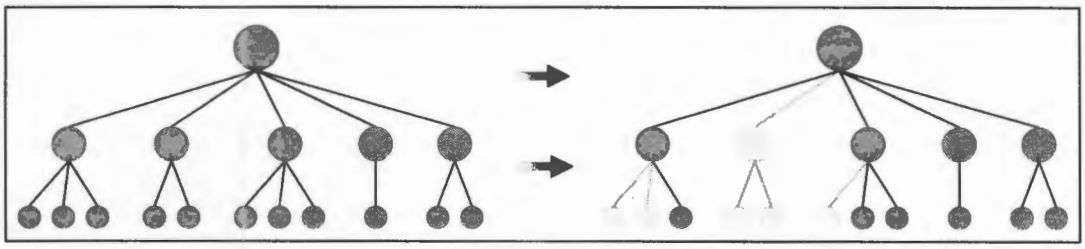


Figure 4.2 - Duplicate octree construction with nodes removed during classification.

In particular this second octree contains the following information:

- The values of voxels in each corner of the volume. Trilinear interpolation may then be performed over the entire sub-volume to more accurately determine the other voxel values.
- The average normal vector of the visible voxels in the sub-volume.
- The average spatial gradient of the visible voxels in the sub-volume.

In the original octree data structure, each node contains an offset into the *voxel-data* area for the raw data of that node as well as offsets into the *octree-data* area for the child nodes. During the incremental rendering process, a node is processed in the following way:

1. Using maximum values and gradients check that this node is visible. If not then recurse back up the octree. (This step is actually carried out in a pre-processing stage, when the classification parameters are modified.)
2. Check that the child node offsets do not reference data which has not yet arrived. If they do then render this node using the partial information contained inside it and its equivalent node in the second octree.
3. Check that *voxel-data* offset references data which has completely arrived for this node. If all the raw data is not available then render the node incrementally.
4. Render the node completely using the raw voxel data.

4.3 Parallel Projection Rendering

The simpler case of the two possible projection matrices is the parallel projection matrix, so we will cover this first. Much of the theory and functionality of the algorithm presented here will carry over to the perspective projection case.

4.3.1 Mathematics of the Factorisation

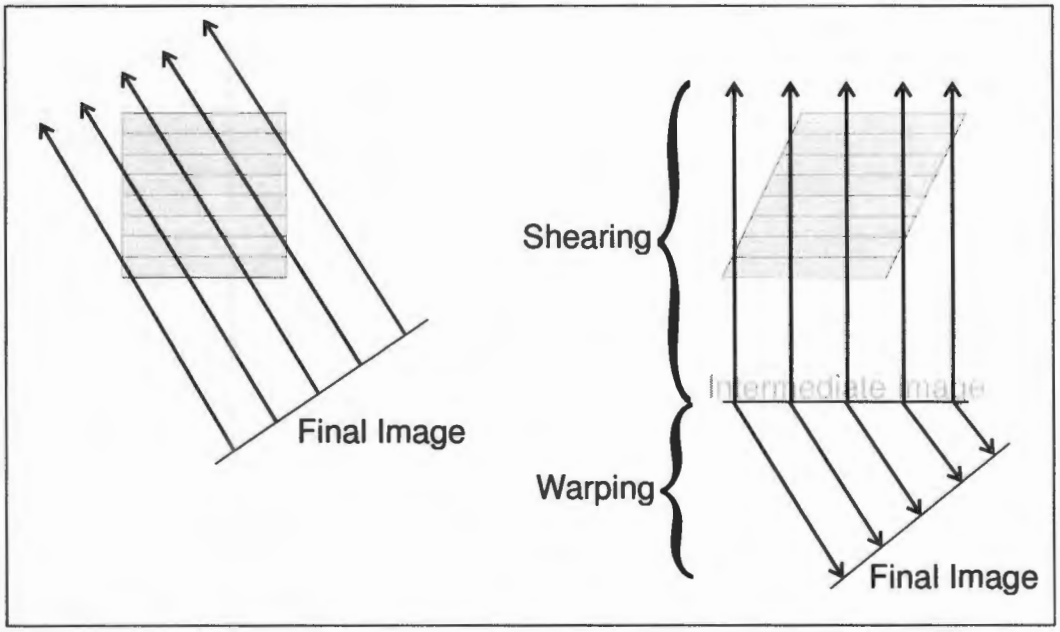


Figure 4.3 - Parallel factorization in two dimensions.

Figure 4.3, depicts a two dimensional equivalent of the Shear-Warp Factorisation of the parallel viewing matrix. This factorisation may be expressed as

$$M_{par} = M_{warp} M_{shear} P$$

where M_{par} is a parallel projection matrix, M_{warp} is a two-dimensional affine warp matrix, M_{shear} is a three-dimensional shear matrix which shears in two directions only, and P is a permutation matrix which ensures that the Z axis is always the primary viewing axis.

A parallel viewing matrix is of the form:

$$M_{par} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Where the a values are combinations of shears, scales, and rotations, while the b values are translations. Without loss of generality it is assumed that these transformations are such that the image is projected onto a plane which sits on the $Z=0$ plane, and is centred around the origin. (If this were not the case, then a multiplication by a combination of scales, translations, and rotations this will be achieved.)

With this assumption it is clear that the Z co-ordinate of any projected point is going to be 0 (in order to lie on the image plane). Thus,

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} M_{par} = \begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix}$$

for any values x, y, and z. The only way this could then be true was if the values a_{31}, a_{32}, a_{33}, and b_3 are all 0. Thus (right-hand multiplication is assumed),

$$M_{par} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

It is also known that the M_{shear} matrix must be of the form,

$$M_{shear} = \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where s_x and s_y are the shear coefficients for the x and y directions respectively. The M_{warp} matrix (a two-dimensional affine warp matrix operating on an image centred at the origin and lying on the $Z=0$ plane) is also known to be of the form (assuming an affine matrix),

$$M_{warp} = \begin{pmatrix} w_{11} & w_{12} & 0 & p_1 \\ w_{21} & w_{22} & 0 & p_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where the w values are combinations of scales, shears, and rotations, and the p values are translations.

Without any loss of generality the permutation matrix P may be assumed to be the identity matrix, therefore,

$$M_{warp} M_{shear} = M_{par}$$

Expanding this out gives,

$$\begin{pmatrix} w_{11} & w_{12} & 0 & p_1 \\ w_{21} & w_{22} & 0 & p_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\therefore \begin{pmatrix} w_{11} & w_{12} & (s_x w_{11} + s_y w_{12}) & p_1 \\ w_{21} & w_{22} & (s_x w_{21} + s_y w_{22}) & p_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

which gives the following solutions immediately:

$$w_{11} = a_{11} \qquad w_{12} = a_{12}$$

$$w_{21} = a_{21} \qquad w_{22} = a_{22}$$

Solving for the s values then gives:

$$\boxed{s_y = \frac{(a_{21}a_{13} - a_{23}a_{11})}{(a_{21}a_{12} - a_{11}a_{22})}}$$

$$\boxed{s_x = \frac{(a_{12}a_{23} - a_{22}a_{13})}{(a_{21}a_{12} - a_{11}a_{22})}}$$

4.3.2 Traversal

The introduction to this chapter (§ 4.1) outlined the problem of traversing the octree in such a way that the occlusion of the sub-volumes (represented by nodes in the octree) is maintained, and also so front-to-back compositing can be used. This problem is similar to that experienced during occlusion compatible traversal of height fields. Anderson [55] presented methods for hidden line and surface removal during rendering of height fields.

Figure 4.4 depicts a two-dimensional version of this problem using a quadtree in place of an octree. The numbers in each case represent the most desirable order in which to visit the nodes such that the above conditions are satisfied. The direction of shear is therefore completely what determines the order of traversal. For the case when the shear is zero (middle quadtree) either traversal may be used.

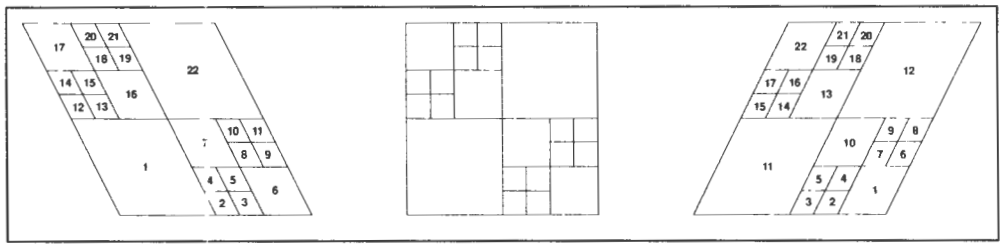


Figure 4.4 - Quadtree traversal orders for parallel projections.

On examination of the orders at each level of the octree, it is apparent that the order at each level is the same as that above it. Therefore it is sufficient to calculate the order of traversal once before rendering based on the shearing factors for the entire volume, and then to use that order recursively at each level of the octree.

In the two-dimensional case above, there are two possible traversal orders based on whether the shear is negative or positive. In the three-dimensional case there are two shears which are independent to one another, so there are then four possible combinations of shears (positive-positive, positive-negative, negative-positive, negative-negative) which results in the choice of four possible traversal orders.

As mentioned above a permutation matrix P is used to ensure that the Z axis is the primary viewing axis. Depending on the axis chosen the order of traversal will obviously be different. So considering that there a three possible viewing axes and there may be four different shearing combinations for each one, there is a total of 12 possible octree traversal orders. In all cases the primary axis and shearing directions are known a-priori, so the traversal order can simply be chosen once and then used throughout the octree.

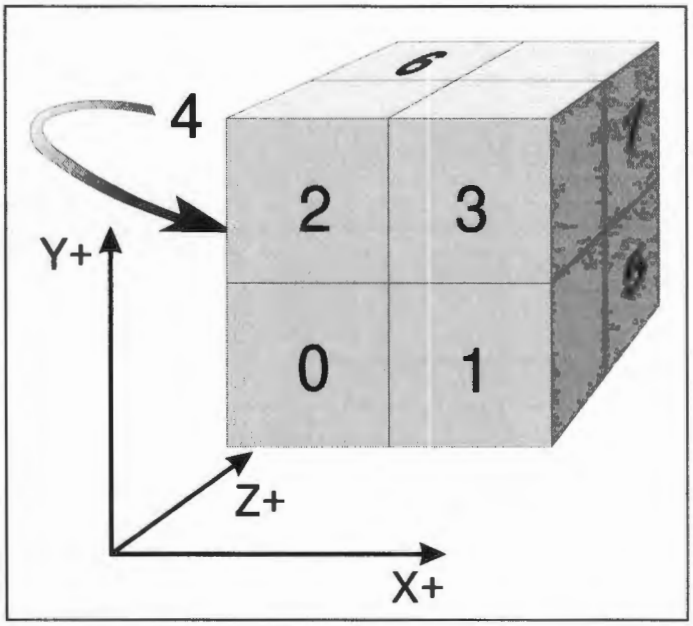


Figure 4.5 - Octree sub-node numbering

The traversal orders are given in the table below where the nodes of the octree are numbered as in Figure 4.5.

X Axis	$S_x<0$, $S_y<0$	0	2	4	6	1	3	5	7
	$S_x>0$, $S_y<0$	2	0	6	4	3	1	7	5
	$S_x<0$, $S_y>0$	4	6	0	2	5	7	1	3
	$S_x>0$, $S_y>0$	6	4	2	0	7	5	3	1
Y Axis	$S_x<0$, $S_y<0$	4	0	1	5	2	6	3	7
	$S_x>0$, $S_y<0$	0	4	5	1	6	2	7	3
	$S_x<0$, $S_y>0$	1	5	4	0	3	7	2	6
	$S_x>0$, $S_y>0$	5	1	0	4	7	3	6	2
Z Axis	$S_x<0$, $S_y<0$	0	1	2	3	4	5	6	7
	$S_x>0$, $S_y<0$	1	0	3	2	5	4	7	6
	$S_x<0$, $S_y>0$	2	3	0	1	6	7	4	5
	$S_x>0$, $S_y>0$	3	2	1	0	7	6	5	4

One of the main advantages of the octree algorithm over the original Shear-Warp algorithm which used RLE coding, is the lack of the need to store three transposed copies of the volume each corresponding to one of the primary viewing directions. This is true for the octree algorithm as a different traversal order is all that is really necessary to cater for a different viewing axis. It is worth noting however that the raw voxel data contained at each leaf-node is still stored in Z-Y-X order, and so needs to be stepped through in a different order depending on the viewing direction. The algorithm presented below chooses to overcome this problem by pre-calculating (at the same time as the traversal order above is calculated) fixed moduli for stepping through the data at each octree level. (The maintenance of a different modulus for each level of the octree is due to the different size of the sub-volumes at each level.)

4.3.3 Trilinear Interpolation

The introduction to this chapter mentioned one of the main problems as being how to render the volume when not all the volume data is present. Due to the hierarchical nature of an octree this problem reduces to rendering specific sub-volumes of the octree using approximate parameters which are stored in the corresponding node.

As mentioned before, the octree node data structure contains the values of the voxels in each corner of the sub-volume, in order to allow trilinear interpolation of voxel values throughout the sub-volume.

It is also necessary to achieve this interpolation in as short a time as possible. A fast incremental version of the algorithm is thus used.

The standard formula for trilinear interpolation is,

$$V(x,y,z)=\left(1-\frac{x}{s}\right)\left(1-\frac{y}{s}\right)\left(1-\frac{z}{s}\right)v_0+\frac{x}{s}\left(1-\frac{y}{s}\right)\left(1-\frac{z}{s}\right)v_1+\left(1-\frac{x}{s}\right)\frac{y}{s}\left(1-\frac{z}{s}\right)v_2+$$
$$\left(1-\frac{x}{s}\right)\left(1-\frac{y}{s}\right)\frac{z}{s}v_3+\frac{x}{s}\frac{y}{s}\left(1-\frac{z}{s}\right)v_4+\frac{x}{s}\left(1-\frac{y}{s}\right)\frac{z}{s}v_5+\left(1-\frac{x}{s}\right)\frac{y}{s}\frac{z}{s}v_6+\frac{x}{s}\frac{y}{s}\frac{z}{s}v_7$$

where (x,y,z) is the relative position within the sub-volume and v_n (where n=[0,7]) are the values of the voxels at each corner of the sub-volume. By expanding this equation and then collecting together the v_n values one gets,

$$V(x,y,z)=\frac{\left[\begin{array}{l}s^3v_0+s^2x(v_1-v_0)+s^2y(v_2-v_0)+s^2z(v_3-v_0)+sxy(v_0-v_1-v_2+v_4)+\\ sxz(v_0-v_1-v_3+v_5)+syz(v_0-v_2-v_3+v_6)+xyz(v_7-v_6-v_5-v_4+v_3+v_2+v_1-v_0)\end{array}\right]}{s^3}$$

Then by identifying constants,

$$V(x,y,z)=C_0+xC_1+yC_2+zC_3+xyC_4+xzC_5+yzC_6+xyzC_7$$

where the C values are constants.

Inside the algorithm this function is evaluated inside a 3rd order loop , each level of which corresponds to an unknown in the above equation. So by using a process of loop-unrolling, this function can be evaluated at every point (incrementally) by just using additions and a small amount of initial calculations.

4.3.4 Algorithm

The pseudo-code for the algorithm is presented below with descriptions of the algorithm.

```
PROC RenderParallelVolume(root_node,projection_matrix)

    (viewing_axis, shear_x, shear_y, warp_matrix) =
                   CalculateViewingParameters(projection_matrix);

    (traversal_order) = CalculateTraversalOrders(viewing_axis,
                                                 shear_x,
                                                 shear_y);

    RenderParallelNode(root_node, traversal_order);

    WarpImage(warp_matrix);
END
```

RenderParallelVolume calculates all viewing parameters, then builds an array of node numbers according to the traversal order. *CalculateViewingParameters* performs a factorization of the projection matrix, and computes the primary viewing and shearing directions. The warp matrix is also computed. The *RenderParallelVolume* function then begins a recursive rendering of the octree to an intermediate compositing image. Finally this compositing image is warped into the final image.

```
PROC CalculateTraversalOrders(viewing_axis, shear_x, shear_y)
```

```
    CASE (viewing_axis)
        X   :   IF (shear_x<0)
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[0]
                    ELSE
                        RETURN TraversalOrderTable[2]
                    END
                ELSE
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[1]
                    ELSE
                        RETURN TraversalOrderTable[3]
                    END
                END

        Y   :   IF (shear_x<0)
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[4]
                    ELSE
                        RETURN TraversalOrderTable[6]
                    END
                ELSE
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[5]
                    ELSE
                        RETURN TraversalOrderTable[7]
                    END
                END

        Z   :   IF (shear_x<0)
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[8]
                    ELSE
                        RETURN TraversalOrderTable[10]
                    END
                ELSE
                    IF (shear_y<0)
                        RETURN TraversalOrderTable[9]
                    ELSE
                        RETURN TraversalOrderTable[11]
                    END
                END
    END
END
```

The *CalculateTraversalOrder* function uses the primary viewing direction and the shearing factors to calculate an ordered list of nodes used for the traversal order of the octree.

```
PROC RenderParallelNode(root_node, traversal_order)

    FOR(i = 1 TO 8)
        child_info = root_node.child[traversal_order[i]];

        IF (IsVisible(child_info))
            IF (DataPresent(child_info))
                IF (IsLeafNode(child_info))
                    PerformRendering(child_info);
                ELSE
                    RenderParallelNode(child_info.node,
                                       traversal_order);
                END
            ELSE
                PerformPartialRendering(child_info);
            END
        END
    END
```

```
END
```

RenderParallelNode is a recursive function which attempts to render the volume represented by the octree by traversing the octree to a maximum depth. The function accepts a node in the octree, and it loops through all of its children. For each of the child nodes:

If it is visible and the node is a leaf node with its data present, then it is rendered normally.

If it is visible and the node is a leaf node but its data is not present then it is rendered using the partial rendering function.

If it is visible and the node is not a leaf node and the data for the child nodes are present then the *RenderParallelNode* function is called with this child node.

If it is visible and the node is not a leaf node but the data for the child nodes is not present, then the node is rendered using the partial rendering function.

```
PROC PerformRendering(node_info)

    FOR(z = 0 TO node_info.size)

        u = node_info.x + (node_info.z + z) * shear_x;
        v = node_info.y + (node_info.z + z) * shear_y;

        FOR(y = 0 TO node_info.size)

            FOR(x = 0 TO node_info.size)

                IF (IsOpaque(u + x, v + y))
                    x = x + SkipOpaquePixels(u + x, u + y) - 1;
                ELSE
                    (colour, alpha) = CalculatePixel(node_info.data[x,y,z]);

                    CompositePixel(u + x, v + y, (colour,alpha));
                END
            END
        END
    END
END
```

PerformRendering renders the specified octree node using the raw data referenced in that node. The function loops through the node and composites the slices onto the intermediate image. During an individual scan-line runs of pixels (voxels) may be skipped if they are already opaque in the intermediate image. For non-opaque pixels the corresponding voxel is retrieved from the raw data. (It should be noted here that a variable modulus has to be used to reference this data as a 3 dimensional array, and this modulus would be calculated at the same stage as the traversal order.) Then using the *CalculatePixel* function (which uses shade tables) the shading of that voxel is calculated. The resulting shaded pixel is then composited into the intermediate image.

```
PROC PerformPartialRendering(node_info)

    FOR(z = 0 TO node_info.size)

        u = node_info.x + (node_info.z + z) * shear_x;
        v = node_info.y + (node_info.z + z) * shear_y;
```

```
        FOR(y = 0 TO node_info.size)

            FOR(x = 0 TO node_info.size)

                IF (IsOpaque(u + x, v + y))
                    x = x + SkipOpaquePixels(u + x, u + y) - 1;
                ELSE
                    voxel =
                     TrilinearInterpolate(node_info.corner_values,
                                          node_info.average_normal,
                                          node_info.average_gradient,
                                          node_info.size,
                                          x, y, z);

                    (colour, alpha) = CalculatePixel(voxel);

                    CompositePixel(u + x, v + y, colour, alpha);
                END
            END
        END
    END
END
```

PerformPartialRendering renders a partial octree node by using trilinear interpolation of the values throughout the sub-volume. The compositing is performed in exactly the same fashion as the *PerformRendering* function but the calculation of the voxel relies on the *TrilinearInterpolate* function. This function uses the voxel values at each corner of the sub-volume, the average normal of opaque voxels in the volume, and the average gradient of opaque voxels in the volume to determine a particular voxel value. (Note: In the actual implementation, and as mentioned in a previous section, a incremental version of the trilinear interpolation function is used to rapidly calculate the voxel values.)

```
PROC CompositePixel(comp_x, comp_y, colour, alpha)

    IF (alpha > TRANSPARENT_VALUE)
        comp_image[comp_x,comp_y] = comp_image[comp_x,comp_y] OVER
                                    (colour,alpha);

        IF (comp_image[comp_x,comp_y].alpha >= OPAQUE_VALUE)
            comp_flags[comp_x,comp_y] = TRUE;
        END
    END
END
```

This function composites a pixel into the compositing image (intermediate image) using the OVER operator. If the resultant pixel becomes completely opaque then that pixel is flagged as being opaque and will then be skipped in future passes.

```
PROC SkipOpaquePixels(comp_x, comp_y)

    i = 0;

    WHILE (EndOfScanline(comp_x + i, comp_y))

        IF (comp_flags[comp_x + i, comp_y])
            i = i + 1;
        ELSE
            RETURN i;
        END
```

```
    END

    RETURN i;
END
```

SkipOpaquePixels calculates the length of a run of opaque pixels in the compositing image. In the original Shear-Warp a algorithm used forests of trees (implemented as offsets in each pixel) to allow this skipping to be efficient). In this implementation bit flags are used for each pixel and rapid bit manipulation functions are used to scan runs. (For long runs, on a 32-bit processor, 32 pixels may be skipped in one machine instruction.) We found that this approach allowed for simpler and more flexible coding as well as slightly improving the efficiency.

```
PROC IsOpaque(comp_x, comp_y)

    RETURN comp_flags[comp_x, comp_y];
END
```

This returns whether the specified pixel in the intermediate image is opaque or not.

4.4 Perspective Projection Rendering

The perspective projection case is a lot more difficult to handle as the viewing rays are no longer parallel, which implies scaling of the data with distance, and also that the octree traversal may be different in different areas of the volume. A new approach has been developed to handle the scaling problem as efficiently as possible and this is presented in the section *Scaling of* Slices below. The partial node rendering technique using trilinear interpolation remains exactly the same as that used in the parallel case.

4.4.1 Mathematics of the Factorisation

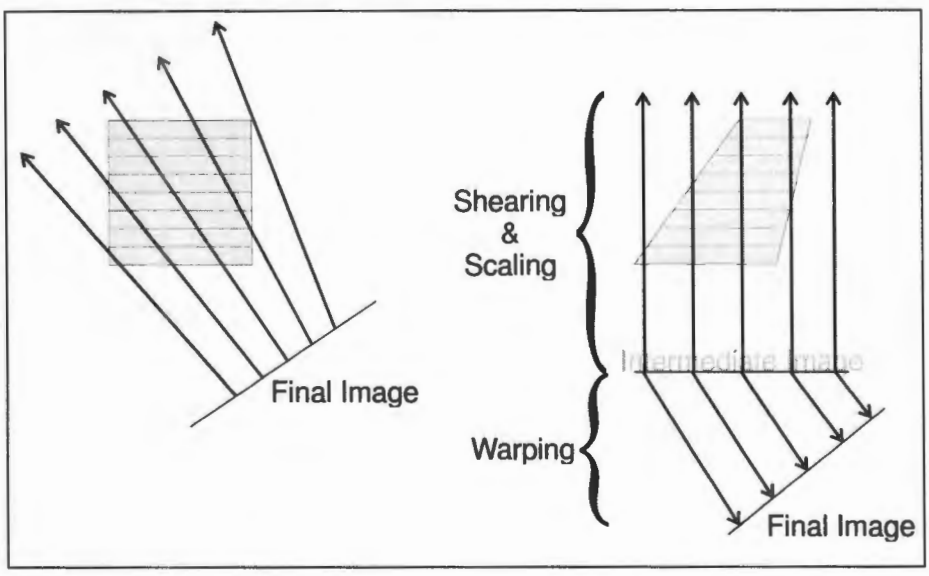


Figure 4.6 - Perspective factorization in two dimensions.

Figure 4.6, depicts a two dimensional equivalent of the Shear-Warp Factorisation of the perspective viewing matrix. This factorisation may be expressed as

$$M_{prsp} = M_{warp} M_{scale-shear} P$$

where M_{prsp} is a perspective projection matrix, M_{warp} is a two-dimensional perspective warp matrix, $M_{scale-shear}$ is a three-dimensional scale and shear matrix which shears in two directions only and scales in the other direction, and P is a permutation matrix which ensures that the Z axis is always the primary viewing axis.

The parallel projection matrix, when transformed by three dimensional affine transformation matrices, remains a parallel projection matrix. i.e. The 4th row of the matrix remains $\begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$. However when a perspective projection matrix is transformed by three dimensional affine transformation matrices, then it becomes a perspective transformation matrix, of the form:

$$M_{prsp} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix}$$

This matrix is thus our perspective viewing matrix, which has to be factorised. Without loss of generality it is assumed that this transformation is such that the image is projected onto a plane which sits on the $Z=0$ plane, and is centred around the origin. (If this were not the case, then a multiplication by a combination of scales, translations, and rotations this will be achieved.)

With this assumption it is clear that the Z co-ordinate of any projected point is going to be 0 (in order to lie on the image plane). Thus,

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} M_{par} = \begin{pmatrix} x' \\ y' \\ 0 \\ 1 \end{pmatrix}$$

for any values x, y, and z. The only way this could then be true was if the values a_{31}, a_{32}, a_{33}, and b_3 are all 0. Thus (right-hand multiplication is assumed),

$$M_{prsp} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix}$$

It is also known that the $M_{scale-shear}$ matrix must be of the form,

$$M_{scale-shear} = \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & q & 1 \end{pmatrix}$$

where s_x and s_y are the shear coefficients for the x and y directions respectively, and q is a scaling factor.

Unlike the parallel projection case (where the warp matrix was affine) the resulting warp matrix for the perspective projection case is a perspective warp matrix of the form,

$$M_{warp} = \begin{pmatrix} w_{11} & w_{12} & 0 & w_{13} \\ w_{21} & w_{22} & 0 & w_{23} \\ 0 & 0 & 0 & 0 \\ w_{31} & w_{32} & 0 & 1 \end{pmatrix}$$

which is a non-linear image warping matrix. The values in the third row and third column can actually be any value at all as they are dropped when this matrix is converted to a two dimensional transformation matrix. We assume these values are 0 in order to make the calculations simpler.

Without any loss of generality the permutation matrix P may be assumed to be the identity matrix (see note below), therefore,

$$M_{warp} M_{scale-shear} = M_{prsp}$$

Expanding this out gives,

$$\begin{pmatrix} w_{11} & w_{12} & 0 & w_{13} \\ w_{21} & w_{22} & 0 & w_{23} \\ 0 & 0 & 0 & 0 \\ w_{31} & w_{32} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & s_x & 0 \\ 0 & 1 & s_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & q & 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ 0 & 0 & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix}$$

$$\therefore \begin{pmatrix} w_{11} & w_{12} & (s_x w_{11} + s_y w_{12} + q w_{13}) & w_{13} \\ w_{21} & w_{22} & (s_x w_{21} + s_y w_{22} + q w_{23}) & w_{23} \\ 0 & 0 & 0 & 0 \\ w_{31} & w_{32} & (s_x w_{31} + s_y w_{32} + q) & 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ & 0 & 0 & 0 \\ a_{41} & a_{42} & a_{43} & 1 \end{pmatrix}$$

which gives the following solutions immediately:

$$w_{11} = a_{11} \qquad w_{12} = a_{12}$$
$$w_{21} = a_{21} \qquad w_{22} = a_{22}$$
$$w_{31} = a_{41} \qquad w_{32} = a_{42}$$
$$w_{13} = a_{14} \qquad w_{23} = a_{24}$$

Then by solving the equations,

$$s_x a_{11} + s_y a_{12} + q a_{14} = a_{13}$$

$$s_x a_{21} + s_y a_{22} + q a_{24} = a_{23}$$

$$s_x a_{41} + s_y a_{42} + q = a_{43}$$

for the shearing coefficients s_x and s_y, and the scaling coefficient q, the following solutions appear:

$$\boxed{s_x = \frac{a_{13}a_{22} - a_{12}a_{23} + a_{14}a_{23}a_{42} - a_{13}a_{24}a_{42} - a_{14}a_{22}a_{43} + a_{12}a_{24}a_{43}}{a_{11}a_{22} - a_{12}a_{21} - a_{14}a_{22}a_{41} + a_{12}a_{24}a_{41} + a_{14}a_{21}a_{42} - a_{11}a_{24}a_{42}}}$$

$$\boxed{s_y = \frac{a_{11}a_{23} - a_{13}a_{21} - a_{14}a_{23}a_{41} + a_{13}a_{24}a_{41} + a_{14}a_{21}a_{43} - a_{11}a_{24}a_{43}}{a_{11}a_{22} - a_{12}a_{21} - a_{14}a_{22}a_{41} + a_{12}a_{24}a_{41} + a_{14}a_{21}a_{42} - a_{11}a_{24}a_{42}}}$$

$$\boxed{q = \frac{a_{12}a_{23}a_{41} - a_{13}a_{22}a_{41} + a_{13}a_{21}a_{42} - a_{11}a_{23}a_{42} - a_{12}a_{21}a_{43} + a_{11}a_{22}a_{43}}{a_{11}a_{22} - a_{12}a_{21} - a_{14}a_{22}a_{41} + a_{12}a_{24}a_{41} + a_{14}a_{21}a_{42} - a_{11}a_{24}a_{42}}}$$

The perspective warp matrix may then be calculated by using the formula:

$$\boxed{M_{warp} = M_{prsp} M^{-1}_{scale-shear}}$$

as the shearing and scaling matrix is always invertible.

NOTE: The algorithm and theory for the perspective Shear-Warp Factorisation presented here does not work in every possible case. When the eye either approaches the volume very closely, lies inside the volume, or lies exactly on the sides of the volume, these equations will break down. The reason for this is that there is now more than one primary viewing direction, and in order to handle this, the volume will need to be broken up into sections each with its own viewing direction. Each of these sections would then be rendered separately, and the intermediate images would need to be composited together to get the final image. The algorithm presented here chooses rather to disallow these conditions for the sake of simpler implementation and speed of rendering.

4.4.2 Traversal

As with the parallel rendering algorithm, the traversal of the octree has to calculated in such a way to ensure correct occlusion. However in the perspective case this becomes much more difficult to solve as the traversal can be different in different areas of the volume due to the diverging viewing rays. Figure 4.7 depicts three cases of the perspective traversal problem for a quadtree where each of the traversal orders are different but the shearing factors have the same signs.

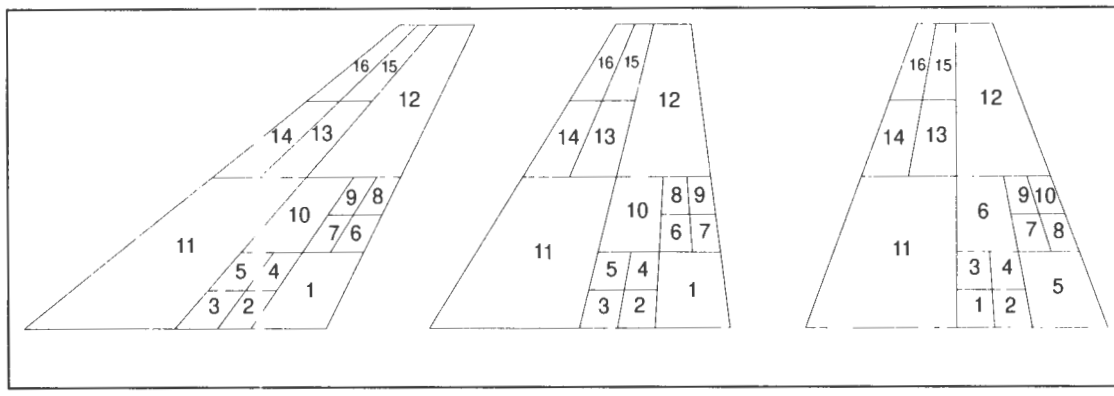


Figure 4.7 - **Quadtree traversal orders for perspective projection.**

On examination of Figure 4.7 it becomes apparent that in order to predict a particular nodes order, the slope of its centre line must be calculated. Then depending on whether this slope is negative or positive (in the X direction) a different traversal is chosen.

As in the parallel case this extends to an octree where a choice must be made between two traversals for each shearing direction. (i.e. Four traversal orders can be chosen from.)

Given any node in the octree with position ***(x,y,z)*** in object space, shearing factors s_x and s_y, sub-volume size v, and a scaling factor q,

$$\begin{aligned} Slope_X &= zs_x(1+qz)(x+v) - (z+v)s_x[1+q(z+v)](x+v) \\ &= (x+v)s_x[z+qz^2 - z - v - zq(z+v) - vq(z+v)] \\ &= -(x+v)s_x[v+zv+vqz+v^2q] \end{aligned}$$

We know that the x and v values will always be positive, and we only need to preserve the sign of this equation so,

$$\boxed{Slope_X = -s_x(1+z+qz+vq)}$$

Similarly for the Y axis,

$$\boxed{Slope_Y = -s_y(1+z+qz+vq)}$$

The choice of traversal orders still remains the same as those presented in the table in the parallel section above.

4.4.3 Scaling of Slices

In the original Shear-Warp algorithm, the volume is scaled such that the slice closest to the viewer is scaled by a factor of 1 and the slices behind it are scaled by a factor <1. Then during the compositing of slices which are behind the first slice, the voxels are averaged together and then composited. This averaging would of course cover more voxels as the slices progress towards the rear of the volume.

However in our octree algorithm (as mentioned before) there is a difficulty in filtering neighbouring voxels, due to the structure of the octree. Before, the filtering was omitted and the results were found to be acceptable, however in this case that solution would lead to very noticeable aliasing artifacts.

The algorithm presented here chooses to overcome this problem by instead scaling the volume such that the slice furthest from the viewer is scaled by a factor of 1, and all slices closer to the viewing are scaled by a factor >1. This of course puts some limitations on the range of the scaling factor, as the intermediate image could become exponentially large for very large scaling factors, but this was not found to be a problem. For any acceptable perspective image the scaling factors do not result in a very large intermediate image.

Due to the fact that the slices are only ever scaled up, omitting the filtering step becomes a lot more acceptable as the aliasing artifacts are vastly reduced and no information is lost. Once all the slices have been composited into the intermediate image, the entire image then simply needs to be scaled down. However there already exists an perspective warp matrix which will operate on this image, so this matrix is then just multiplied by a another 2 dimensional scaling matrix before the rendering. The other advantage of having the warp matrix perform this scaling is that it will perform filtering on the image, thus further reducing the aliasing artifacts produced by the earlier omission of filtering.

4.4.4 Algorithm

The pseudo-code for the algorithm is presented below along with descriptions of each of the functions.

```
PROC RenderPerspectiveVolume(root_node,projection_matrix)

    (viewing_axis, shear_x, shear_y, scale, warp_matrix) =
                          CalculateViewingParameters(projection_matrix);

    (traversal_orders) = PrepareTraversalOrders(viewing_axis);

    RenderPerspectiveNode(root_node, traversal_orders);

    WarpImage(warp_matrix);
END
```

The function *RenderPerspectiveVolume* calculates all viewing parameters, then builds a list of arrays of node numbers according to the primary viewing direction. *CalculateViewingParameters* performs a factorisation of the projection matrix to obtain the primary viewing direction, the shearing directions, and the scaling factor. The warp matrix is also computed. The *RenderPerspectiveVolume* function then begins a recursive rendering of the octree to an intermediate compositing image. Finally this compositing image is warped into the final image.

```
PROC PrepareTraversalOrders(viewing_axis)

    CASE (viewing_axis)
        X   :   RETURN TraversalOrderTable[0];

        Y   :   RETURN TraversalOrderTable[4];

        Z   :   RETURN TraversalOrderTable[8];
    END
END
```

By using the primary viewing direction *PrepareTraversalOrders* selects an ordered list of node arrays for the choices of traversal order during rendering. Once the initial viewing direction is known then there is a choice of 4 possible traversal orders depending on the position in the volume.

```
PROC RenderPerspectiveNode(root_node, traversal_orders)

    (dir_x,dir_y) = CalculateSlopes(root_node);

    IF (dir_x<0)
        IF (dir_y<0)
            traversal_order = traversal_orders[0];
        ELSE
            traversal_order = traversal_orders[2];
        END
    ELSE
        IF (dir_y<0)
            traversal_order = traversal_orders[1];
        ELSE
            traversal_order = traversal_orders[3];
        END
    END

    FOR(i = 1 TO 8)
        child_info = root_node.child[traversal_order[i]];

        IF (IsVisible(child_info))
            (IF DataPresent(child_info))
                IF (IsLeafNode(child_info))
                    PerformRendering(child_info)
                ELSE
                    RenderPerspectiveNode(child_info.node)
                END
            ELSE
                PerformPartialRendering(child_info);
            END
        END
    END
END
```

RenderPerspectiveNode is a recursive function which attempts to render the volume represented by the octree by traversing the octree to a maximum depth. The function accepts a node in the octree, and calculates the necessary traversal order using the formula presented in the previous section. It then loops through all of its children and for each of the child nodes:

If it is visible and the node is a leaf node with its data present, then it is rendered normally.

If it is visible and the node is a leaf node but its data is not present then it is rendered using the partial rendering function.

If it is visible and the node is not a leaf node and the data for the child nodes are present then the *RenderPerspectiveNode* function is called with this child node.

If it is visible and the node is not a leaf node but the data for the child nodes is not present, then the node is rendered using the partial rendering function.

```
PROC PerformRendering(node_info)
```

```
    FOR(z = 0 TO node_info.size)

        u = node_info.x + (node_info.z + z) * shear_x;
        v = node_info.y + (node_info.z + z) * shear_y;

        FOR(y = 0 TO node_info.size)

            FOR(x = 0 TO node_info.size)

                IF (IsOpaque(u + x, v + y, z))
                    x = x + SkipOpaquePixels(u + x, u + y, z) - 1;
                ELSE
                    (colour, alpha) = CalculatePixel(node_info.data[x,y,z]);

                    CompositePixel(u + x,
                                   v + y,
                                   z,
                                   (colour,alpha));
                END
            END
        END
    END
END
```

PerformRendering renders the specified octree node using the raw data referenced in that node. The function loops through the node and composites the slices onto the intermediate image. During an individual scan-line runs of pixels (voxels) may be skipped if they are already opaque in the intermediate image. For non-opaque pixels the corresponding voxel is retrieved from the raw data. (It should be noted here that a variable modulus has to be used to reference this data as a 3 dimensional array, and this modulus would be calculated at the same stage as the traversal order.) Then using the *CalculatePixel* function (which uses shade tables) the shading of that voxel is calculated. The resulting shaded pixel is then composited into the intermediate image.

```
PROC PerformPartialRendering(node_info)

    FOR(z = 0 TO node_info.size)

        u = node_info.x + (node_info.z + z) * shear_x;
        v = node_info.y + (node_info.z + z) * shear_y;

        FOR(y = 0 TO node_info.size)

            FOR(x = 0 TO node_info.size)

                IF (IsOpaque(u + x, v + y))
                    x = x + SkipOpaquePixels(u + x, u + y, z) - 1;
                ELSE
                    voxel =
                     TrilinearInterpolate(node_info.corner_values,
                                          node_info.average_normal,
                                          node_info.average_gradient,
                                          node_info.size,
                                          x, y, z);

                    (colour, alpha) = CalculatePixel(voxel);

                    CompositePixel(u + x, v + y, z, colour, alpha);
                END
            END
        END
    END
END
```

PerformPartialRendering renders a partial octree node by using trilinear interpolation of the values throughout the sub-volume. The compositing is performed in exactly the same fashion as the *PerformRendering* function but the calculation of the voxel relies on the *TrilinearInterpolate* function. This function uses the voxel values at each corner of the sub-volume, the average normal of opaque voxels in the volume, and the average gradient of opaque voxels in the volume to determine a particular voxel value. (Note: In the actual implementation, and as mentioned in a previous section, a incremental version of the trilinear interpolation function is used to rapidly calculate the voxel values.)

```
PROC CompositePixel(comp_x, comp_y, comp_z, colour, alpha)

    IF (alpha > TRANSPARENT_VALUE)
        x_loc = comp_x * (scale * comp_z);
        y_loc = comp_y * (scale * comp_z);

        FOR (i = 0 TO (scale*comp_z - 1))
            FOR (j = 0 TO (scale*comp_z - 1))

                comp_image[x_loc + i, y_loc + j] =
                   comp_image[x_loc + i, y_loc + j] OVER (colour,alpha);

                IF (comp_image[x_loc + i, y_loc + j].alpha >= OPAQUE_VALUE)
                    comp_flags[x_loc + i, y_loc + j] = TRUE;
                END
            END
        END

    END
END
```

The *CompositePixel* function composites a pixel into the a block of pixels in the compositing image (intermediate image) using the OVER operator. The size of the block depends on the slice number which is being processed. If any of the resultant pixels becomes completely opaque then they are flagged as being opaque and will then be skipped in future passes.

```
PROC SkipOpaquePixels(comp_x, comp_y, comp_z)

    x_loc = comp_x * (scale * comp_z);
    y_loc = comp_y * (scale * comp_z);

    i = 0;

    WHILE (EndOfScanline(comp_x + i, comp_y))

        IF (CheckAllFlags(x_loc + i*scale, y_loc, comp_z * scale))
            i = i + 1;
        ELSE
            RETURN i;
        END
    END

    RETURN i;
END
```

SkipOpaquePixels calculates the length of a run of opaque pixels in the compositing image. The size of this run does however have to be a multiple of the block size which is determined by the slice number being rendered. This is to prevent the pixels from overlapping.

```
PROC IsOpaque(comp_x, comp_y, comp_z)

    x_loc = comp_x * (scale * comp_z);
    y_loc = comp_y * (scale * comp_z);

    RETURN CheckAllFlags(x_loc + i*scale, y_loc, comp_z * scale);
END
```

This function returns whether any of the specified pixels in the current block in the intermediate image is opaque or not. Once again the size of this block depends on the slice number being rendered.

```
PROC CheckAllFlags(x, y, size)

    FOR (i = 0 TO (size - 1))
        FOR (j = 0 TO (size - 1))

            IF (NOT comp_flags[x + i, y + j]) RETURN FALSE;
        END
    END

    RETURN TRUE;
END
```

CheckAllFlags checks to see if a block of flags in the intermediate image are all set, which indicates that the entire region is opaque.

4.5 Conclusion

This chapter presented a method of rendering a volumetric dataset which is represented by an octree (i.e. hierarchical) data structure. The method makes use of the basic Shear-Warp algorithm but extends it to make use of octrees instead of the classical RLE data structures. Two separate algorithms were developed, one for the parallel projection of the volume and the other for the perspective projection of the volume.

The octree data structure introduced new capabilities into the algorithm such as the ability to render the volume when only part of the data structure is present. This is achieved through approximation of the missing octree nodes (using trilinear interpolation). This "partial rendering" feature makes the algorithm most suitable for incremental rendering of volume data as it arrives over a slow medium (e.g. network link).

In the parallel rendering case the problem of traversing the octree to ensure correct occlusion was solved using a fairly simple method. The result of rendering the volume during traversal of the octree is that large areas of the volume are omitted rapidly when they are completely empty. The final parallel rendering algorithm offers the same features as the traditional shear-warp except it now does not require three transposed copies of the volume (thus using less memory).

Perspective rendering is achieved through a more advanced algorithm as the problems of octree traversal and slice scaling become a lot more complex. The perspective algorithm solves the traversal order problem by re-computing orders at each tree level, while the slice scaling problem is solved by a unique method which moves the scaling stage to the compositing buffer (rather than doing it in the

volume). Limitations were imposed on the position of the eye point (it may not be on or inside the volume) in the perspective case in order to simplify the algorithm.

The only problem experienced with extending the Shear-Warp algorithm to use octrees is that a slight drop in image quality occurs due to the necessary omission of the bilinear filtering stage during data traversal. This occurs due to the complexity of locating neighbouring voxels at the edge of an octree node. A potential solution to this would be for all the octree nodes to overlap by one voxel, thus allowing rapid referencing of neighbours at the expense of some memory. (See §6.3)

The next chapter will present the approach to the validation of the algorithms and theory presented in this and prior chapters. It will then move on to providing detailed test results and explanations.

Chapter 5

Experimental Results

5.1 Introduction

The acceptance of an algorithm for an efficient incremental volume renderer, which will perform satisfactorily on a wide range of workstations, requires the evaluation of many aspects of the algorithms. Of primary importance are memory usage and rendering performance.

In this chapter, numerous hypotheses will be made about the performance of certain key aspects of the algorithm. After presentation of these hypotheses empirical tests will be presented which support the hypotheses.

Due to the complexities of volume visualisation there are many parameters which may be altered which cause the algorithms to perform in different ways. The empirical tests are thus designed in such a way as to extensively test every possible combination of parameters which will have an effect on: the compression ratio, the rendering performance, and the final image quality.

The results of the tests are presented in graph form in *Appendix C*.

5.2 Hypotheses

The two major hypotheses of this thesis are: that the compression ratios of the octree compressed volumes allow for a major memory reduction during rendering (about 50% is expected); and that this octree structure will facilitate incremental rendering at a reasonably fast rate (under 5 seconds per rendering).

We postulate that the compression ratios obtained with octree compression are roughly equivalent to those achieved through RLE compression. Also by using the octree compressed volume during rendering a run-time memory improvement in excess of 50% can be obtained when compared to using the RLE compressed volume. This enables the algorithms to perform reasonably well on low-end workstations which do not have large primary memories.

Due to the hierarchical nature of the octree, information is coded into the nodes of the octree such that the sub-volumes represented by those nodes may be approximated when the raw-data for the sub-volumes is not available. We postulate that a minimally (only the very basic morphology) recognisable volume will be obtainable with a very small dataset, and that this approximation will improve as more data arrives.

In order for incremental rendering to be effective, the rendering times have to be many orders of magnitude faster than the rate of transmission. (The entire approximated volume has to be continually re-rendered during transmission in order to perceive improvements to the approximations.) Thus assuming an average size compressed volume of 2 Mb, then a network link with a bandwidth of 32K-bits/sec will need 9 minutes to transfer the entire volume. During the 9 minutes of transmission it would be desirable for the volume to be rendered say every 5 to 10 seconds to give an impression of the progress. The algorithms presented in previous chapters will achieve this required rendering performance.

Other more minor hypotheses are:

- Using node compression on leaf-nodes of the octree should be preferable (in terms of performance and compression ratio) to not using node compression.
- Improvements in the performance of octree classification can be gained through the use of a node-cache and that there must be an optimal cache size which should be similar for all standard volumes.
- The parallel rendering performance when using octrees is comparable to the performance when using RLE data structures.
- Perspective rendering of the volume data is supported as well.

A testing methodology will be presented which allows for the testing of these hypotheses using a set of standard volume datasets.

5.3 Test Data

The volumetric datasets which were used to test the algorithms are as follows:

- ***MR Head*** - A magnetic resonance study of a human head, with the skull partially removed to reveal the brain. The volume contains 109 slices each of 256 by 256 points. Each point is a 16-bit integer. The data was captured using a Siemens Magnetom. This data set represents a standard case, with fairly high tissue definition and easily recognisable features. This volume is used extensively in the testing of volume visualisation systems, so by presenting results using this volume, direct comparisons may be made with other systems.
- ***MR Knee*** - A magnetic resonance study of a human knee, from just above the knee joint to just below the joint. The volume contains 127 slices each of 256 by 256 points. Each

point is a 16-bit integer. The data was captured using a Siemens Magnetom. This data set is the largest of all, and also contains a large amount of noise, so the response of the algorithms to noise is tested.

- ***Engine*** - An engine part. The volume contains 110 slices each of 256 by 256 points. Each point is an unsigned 8-bit integer. This volume is very regular in shape, and the surfaces are very clearly defined with barely any noise at all. This volume thus allows us to validate that the algorithms maintain the structural integrity of the volumes, and also to test the correctness of the surface shading. Also the response of the octree representation to regular (Euclidean) surfaces is tested.

- ***CT Head*** - A computed tomography study of a cadaver's head. The volume contains 113 slices each of 256 by 256 points. Each point is a 16-bit integer. The data was captured using a General Electric CT Scanner. This is a fairly noise-free volume, with very good bone surface definition, and is thus a good candidate to test the effectiveness of the compression, as well as the smoothness of the shading on a non-Euclidean surface. Due to the fact that only the bone is of interest, a lot of the head data may be removed, so the compression algorithm's response to this *opportunity* is tested.

The two magnetic resonance volumes, as well as the computed tomography volume were obtained from the University of North Carolina (Chapel Hill), and were captured at the North Carolina Memorial Hospital. The engine dataset was made available by the Stanford Computer Graphics Group at Stanford University in order for users of the original Shear-Warp algorithm to validate the effectiveness of their algorithm.

Before presenting the results it is worth noting that each of the test volumes were first "optimised" such that the resulting raw dataset contained 8-bit unsigned integers, and the range of values 0-255 accurately represented the information content of the volume.

5.4 Methodology

In order to prove the effectiveness of the algorithms presented in this thesis, a volume rendering class library was constructed which implements the various algorithms.

Many of the parameters in the system are completely orthogonal to one another (for example isosurface level, and object rotation), so ranges of test values were devised for them such that every aspect of the algorithms were exercised.

To facilitate methodical and organised statistics, a simple scripting language was developed which allowed pre-set test sequences to be established in test scripts. These test scripts were then executed on different data sets (in order to ensure independence) on a number of workstations. The script language allowed for the following operations:

OCT	Create an instance of an octree compressed volume.
~OCT	Destroy an instance of an octree compressed volume.
RAW	Create an instance of a raw data volume.
~RAW	Destroy an instance of a raw data volume.
RLE	Create an instance of a RLE compressed volume.
~RLE	Destroy an instance of a RLE compressed volume.
OCTLOAD	Load in data from a file into an octree volume object.
OCTSAVE	Save data to a file from an octree volume object.
OCTCONSTRUCT	Construct the data inside an octree volume object from a raw volume object.
OCTCLASSIFY	Perform classification on the data inside an octree volume object.
OCTRENDERPARA	Perform a parallel projection rendering of an octree volume object.
RAWLOAD	Load in data from a file into an raw volume object.
RAWSAVE	Save data to a file from an raw volume object.
RAWOPTIMIZE	Optimise the data contained in a raw volume object.
RAWPARSE	Parse a file of raw data into a raw volume object.
RLELOAD	Load in data from a file into an RLE volume object.
RLESAVE	Save data to a file from an RLE volume object.
RLECONSTRUCT	Construct the data inside an RLE volume object from a raw volume object.
RLECLASSIFY	Perform classification on the data inside an RLE volume object.
RLERENDER	Perform a parallel projection rendering of an RLE volume object.
CLEARLOG	Clear the current log file.
SAVEOCTIMAGE	Save the rendered image obtained from an octree volume object.
SAVERLEIMAGE	Save the rendered image obtained from an RLE volume object.
LOG	Create a new log file.
~LOG	Close the current log file.
MESSAGE	Write a message into the current log file.
OCTRENDERPRSP	Render a perspective image using an octree volume object.
OCTRENDERPARAI	Incrementally render a parallel projection image of an octree volume

object.

OCTRENDERPRSPI Incrementally render a perspective projection image of an octree volume object.

SAVEOCTIMAGEI Incrementally save images. (Used for creation of frames for animations.)

Due to the number of parameters and the ranges through which they vary, the duration of the tests increased geometrically as new parameters were added. Thus the testing was executed on a pair of work-stations running in parallel. Due to the fact that the workstations were not identical a performance drop is to be expected for the algorithm running on the slower workstation. This, of course, prohibits the comparison of performance data between algorithms running on different workstations. It was thus necessary to confine the testing of certain volumes to an individual workstation and then only perform comparisons between the various operations on these volumes. This does not restrict our testing in any way as it is meaningless to compare operations on different volumes.

The workstations used for testing were as follows:

- Silicon Graphics Indy R4600 (64Mb Main Memory; 16 Kb Data Cache; 16Kb Instruction Cache; and 512Kb Secondary Cache) running tests on the "MR Head" and "MR Knee" datasets.
- Silicon Graphics Indy R4000 (48Mb Main Memory; 8Kb Data Cache; 8Kb Instruction Cache; 1Mb Secondary Cache) running tests on the "CT Head" and "Engine" datasets.

In order to reduce the error introduced by other workstation overhead, each of the tests were repeated numerous times and the results were averaged. We begin by testing the initial compression stages of the data volumes, and then move on to testing the classification stages and the rendering stages. Most of the results are presented in graph form in *Appendix C*.

The table below contains a reference to all the tests which were performed:

Initial Compression Stage	**§5.5**	
Octree construction without node compression.	§5.5.1	Measurement of the three stages of octree construction in terms of execution time and resulting compression ratios. Individual octree leaf nodes were not compressed further.
Octree construction with node compression	§5.5.2	Individual octree leaf nodes were compressed using the leaf node compression algorithm. The results are

		compared with those without node compression.
RLE Compression	§5.5.3	The execution time and compression ability of a pure run-length encoding compression algorithm is compared to the results for octree compression.
Classification	**§5.6**	
Determination of optimal cache size	§5.6.1	A set of tests which vary the size of the cache used during octree data classification on a variety of datasets. From this an optimal cache size is selected.
Comparison with RLE	§5.6.2	Measurements of octree classification performance and a comparison with RLE classification.
Rendering (Parallel)	**§5.7.1**	
Full octree rendering	§5.7.1.1	Measurement of standard preparation stages for both RLE rendering and octree rendering. Actual rendering times of volumes using the octree method are given.
Comparison with partial octree rendering	§5.7.1.2	Measurement of rendering times for rendering a "worst-case" approximated volume. The times are compared to full octree rendering of the original data.
Comparison of octree and RLE rendering	§5.7.1.3	Comparison of full octree rendering times and normal RLE rendering times using equivalent datasets.
Rendering (Perspective)	**§5.7.2**	
Full octree rendering	§5.7.2.1	Measurement of initial preparation stages for octree rendering. Actual rendering times of volumes using the octree method are given.
Comparison with partial octree rendering	§5.7.2.2	Measurement of rendering times for rendering a "worst-case" approximated

		volume. The times are compared to full octree rendering of the original data.
Images and Animations	**§5.8**	Sets of images obtained from rendering the various test volumes under set conditions using the various algorithms. The actual images are in *Appendix D*, however this section gives the necessary background for interpreting the images as well as a discussion of the results.

5.5 Performance of Initial Compression

These tests analysed the time taken and memory required for the generation of the initial compressed volumes. This operation would generally be performed once off for a few isosurface levels and then never again, so the performance is not critical, however the effectiveness of the compression is a concern.

Each of the four test volumes were tested and results for both standard octree compression as well as octree compression with leaf node compression are presented. A comparison is made between compression with or without the leaf-node compression, taking into account the necessary partial-decompression of the volume in the latter case. The compression ratios are then compared with those obtained using the RLE compression method.

For the compression of a volume using any form of algorithm the identification of a meaningful range of values is required. The raw data in this case has already been optimised to the range 0 to 255, where 255 represents maximum density. However many of the lower range values may represent empty space or less dense substances which we do not wish to render. Our tests thus vary the lower limit, in each of the cases from 1 up to 200 thus removing more and more data values from the volume at each level. It is worth noting however that this could also have been performed using the upper limit value as well but this would have had the same effect. Generally it is always necessary to set the lower-limit to some non-zero value in order to simply avoid the storage of empty-space.

Both the RLE compressed volumes and the octree volumes were then compressed at the various levels. For the octree volumes however there is an additional parameter in the desired depth of the octree. As the depth of the octree increases the hierarchical representation of the volume becomes progressively more accurate. However for large depths the size of the octree data structure explodes (increases according to $(8^{N+1} - 1)$ where N is the depth), so it is necessary to carefully choose the maximum practical depth. It was found that a depth of 6 was as large as the octree could grow without becoming unmanageable and impractical. At this level each node would represent a 4 by 4 by 4 block of voxels

in each of the test volumes, resulting in a maximum of 2097151 nodes. Thus the depth parameter is varied between 1 and 6 for each of the levels, for the creation of the octree compressed volumes.

5.5.1 Construction without Node Compression

5.5.1.1 Performance

The performance tests covered the three distinct stages of the construction process:

1. Initial construction of octree data structure from the raw data.
2. Reordering of the nodes into breadth-first order.
3. Gathering of raw data pertaining to leaf nodes of the octree.

The graphs in figures **C.1** to **C.4** depict the performance of the octree compression algorithm (without using node compression) over each of the four test volumes. Each vertical section of the graph represents an increasing octree depth (the left most section represents a depth of 1, while the right most section represents a depth of 6). Inside each section the lower limit is varied through the range mentioned above.

We expect that the execution time of all stages of the compression should increase steadily as the depth of the octree increases and as the lower level is increased (i.e. More data is excluded) the execution time should drop.

By analysing the graphs one can pick up the following trends:

- The initial octree construction process is the most expensive of the three stages.
- The initial construction process becomes progressively more expensive as the octree depth increases, but not dramatically. An advantage of no more than a second is gained by using a depth of 3 or 4 over a depth of 6.
- The reordering process has a negligible impact on the performance.
- For very small lower limits the performance is noticeably slower, but as the lower limit increases the algorithm performs better, initially at a logarithmically decreasing rate, then at an almost linear rate.
- If one examines the total times on the graphs one can see that a worst case time of around 6 seconds can be expected, which considering that this process only has to be executed once initially, is very acceptable.

The most surprising aspect of these results is that the total time taken to compress the volume using deeper octrees is not necessarily slower than using a shallower octree. This is good as it means that a deep octree can be used (which can represent the volume closer) without incurring a large overhead.

5.5.1.2 Compression

The results of the volume compression at each of the six octrees depths mentioned above, are presented in figures **C.5** to **C.8**.

In the results of each test case both the compressed volume size is presented as well as the optimal compressed size (calculated by counting the number of opaque voxels in the volume).

The compression of the volumes is expected to increase at an almost linear rate as the depth of the octree increases due to the closer representation of the actual volume. The compression ratios for lower threshold levels are expected to be much lower than higher levels due to the amount of low density material common in volume data. In the case of a very deep octree the compression should approach the optimal limit. (i.e. Only voxels which are above the threshold are stored at all.)

From the graphs it can seen that:

- For greater octree depths the compression rapidly approaches the optimal levels.
- As the lower limit increases from very small values (<40), the improvement in compression is dramatic. This is probably due to the removal of low density values such as those for air. From this level onwards the improvements in compression ratio depend on the nature of the volume.
- The octree compression closely approaches optimal compression for a depth of 6 and a fairly high lower limit.
- From these results one can thus conclude that using a depth of 6 is definitely the most advisable as the compression ratios achieved are vastly superior to those at lower depths, while the time taken for the algorithm to execute is negligibly slower than for other depths.

The results were very much as we expected except that the compression ratio did not come as close to the optimal level (for greater octree depths) as we expected. This indicates that the octree is not closely representing the natural structure of the volume as accurately as we intended, and thus leads us on to considering adding further nodes to the octree method.

5.5.2 Construction with Node Compression

When using node compression each node is simply compressed according to the specified lower-level threshold. The node-compression algorithm makes use of the run-length-encoding scheme, but does however check to see if the resultant node would be larger if compressed (this can happen if a region is very noisy), and then does not compress it. Thus the choice of test volume and threshold level could drastically effect the results.

Once again levels 1 through 200 are calculated, each at octree depths of 1 to 6.

5.5.2.1 Performance

As with the previous section figures **C.9** to **C.12** depict the performance of the octree construction phase, except that the data gathering phase is now replaced with a data compression phase, which further compresses the data in each of the leaf nodes.

We expect similar results to the method of not using node compression. The execution of the first two stages should be identical (as they do not change at all) however the third *gathering* stage is now replaced with a *compression* stage which should take noticeably longer. As the node compression is being performed using RLE compression (which has almost linear complexity) the performance of the compression stage should become better for deeper octrees and higher threshold levels as less data is being compressed. Due to the nature of run-length encoding it is possible for the "compressed" data to be larger than the original. The leaf-node compression algorithm however detects this problem, and will then simply store a particular node as-is, if compression would result in a size increase. This is then more likely to happen as the leaf-nodes cover more complex data better, which is what happens at greater octree depths.

The trends seen in the results are similar to those in the previous section.

- The duration of the algorithm tends to increase with depth, however the increase is not dramatic at all.
- The data compression stage can actually take longer than the octree construction phase for depths less than 3, however once the depth becomes greater the octree construction phase takes longer while the compression is faster.
- The performance of compressing the engine data set increases dramatically at level 140. This is due to the almost discrete nature of the density levels in the engine dataset and the fact that one entire density range is now suddenly being omitted.

These results are exactly as we predicted. The only unexpected feature of the results was that for deeper octree depths the node compression time can be less than a third of the total compression time. This is much faster than was initially expected.

Once this compressed volume has been transmitted to a workstation it is necessary to decompress each of the nodes for the rendering algorithm to function. Figures **C.13** to **C.16** depict the decompression time for each of the test volumes.

As with the compression stage we expected the execution time of the decompression to decrease with increasing octree depth and higher threshold levels.

From the graphs one can see that this is exactly what happens. The worst case decompression time in all the results was still under a second which (considering it only has to performed once) is a very acceptable result. It therefore becomes clear that the use of node compression adds barely a few seconds of total performance drop to the whole compress-transmit-decompress cycle. If this method can then improve the compression noticeably then it is very worthwhile.

5.5.2.2 Compression

The compression ratios achieved by using node compression are presented in the figures **C.17** to **C.20** in the same format as the results for non-compression of the nodes.

We now expect the compression of the volume to be a lot closer to the optimal level when the octree is deeper.

From the results it is noticeable that:

- The achieved compression ratios are almost constant across the various depths now, due to the leaf-node compression.
- Similar features as with pure octree compression are seen for higher depths, such as near-optimal compression for high lower-levels.

As expected the compression ratios are now on average closer to optimal compression, except they are not as close as we would have hoped. The most surprising aspect of the results is that the average compression ratios no longer increase as the octree depth increases. This indicates that any octree depth could then be used and a similar compression ratio obtained. However in a volume rendering system (as mentioned above) it is necessary to decompress these nodes once they arrive at the workstation in which case the data would be compressed according to the compression graphs presented in the previous section. Thus, considering that the node-compression provides an overall improvement in compression, it is still advisable to use a maximum depth with node compression. The performance penalties for selecting these parameters are still negligible considering a worst case compression time of 7.5 seconds and a worst case decompression time of 0.9 seconds.

5.5.3 Comparison with RLE Compression

To compare these previous results with those obtained using RLE compression, the same four volumes were compressed at each of the *lower-threshold* levels. A comparison is drawn below (Figure 5.8 to Figure 5.11) with the octree method using node-compression and a tree depth of 6.

The compression ratios of RLE compression are generally fairly good and so we expected them to be very near the compression ratios achieved using octree compression. However when large areas of the volume are below the threshold the octree algorithm (due to the fact that it works in three dimensions) should represent the "empty space" more efficiently. Thus we expect octree compression to outstrip RLE compression for higher threshold levels.

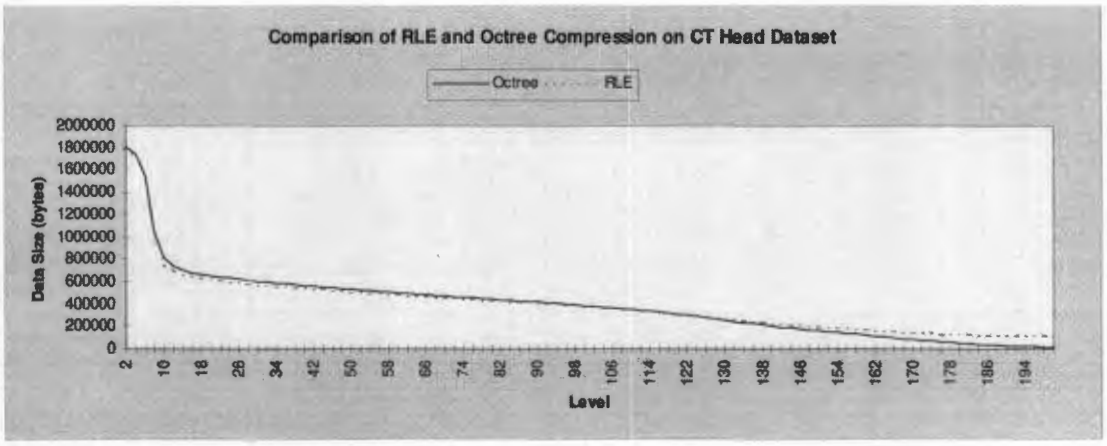


Figure 5.8 - Octree vs. RLE compression levels for the CT Head dataset.

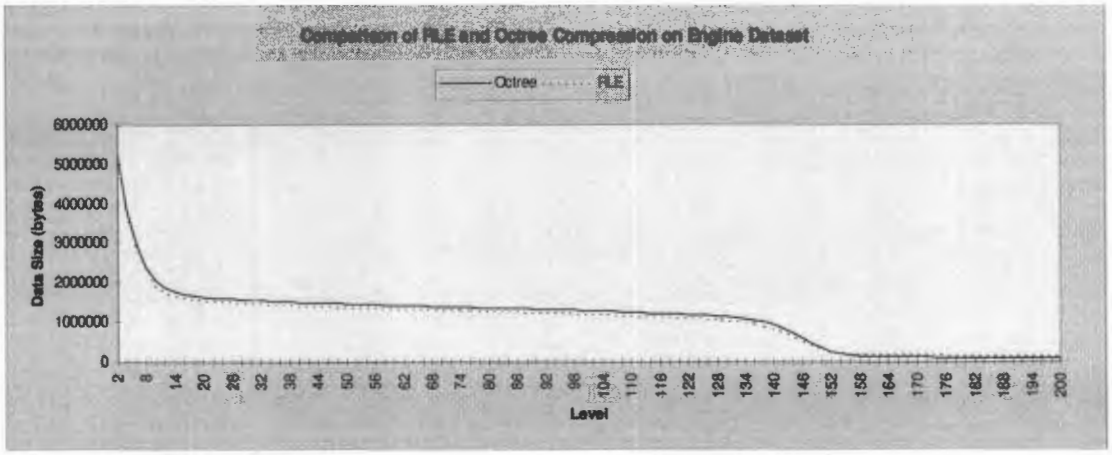


Figure 5.9 - Octree vs. RLE compression levels for the Engine dataset.

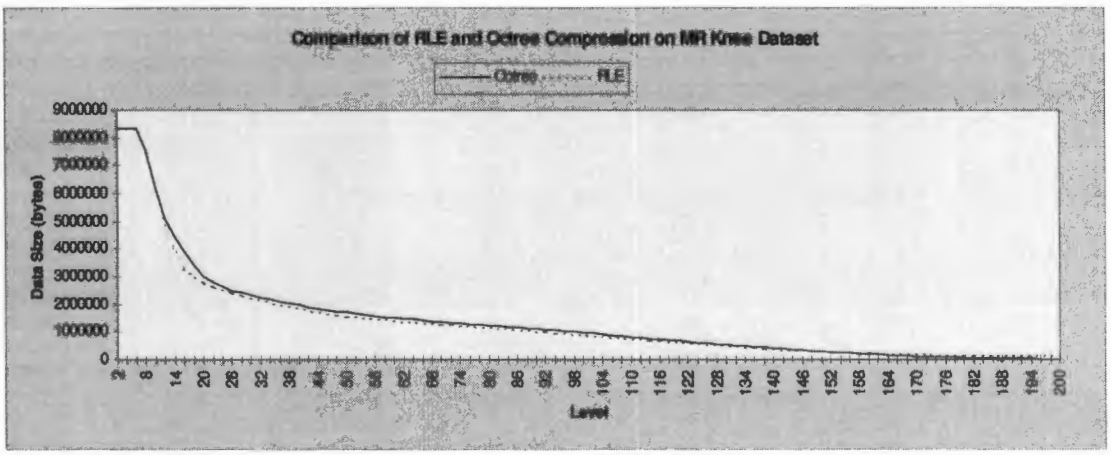


Figure 5.10 - Octree vs. RLE compression levels for the MR Knee dataset.

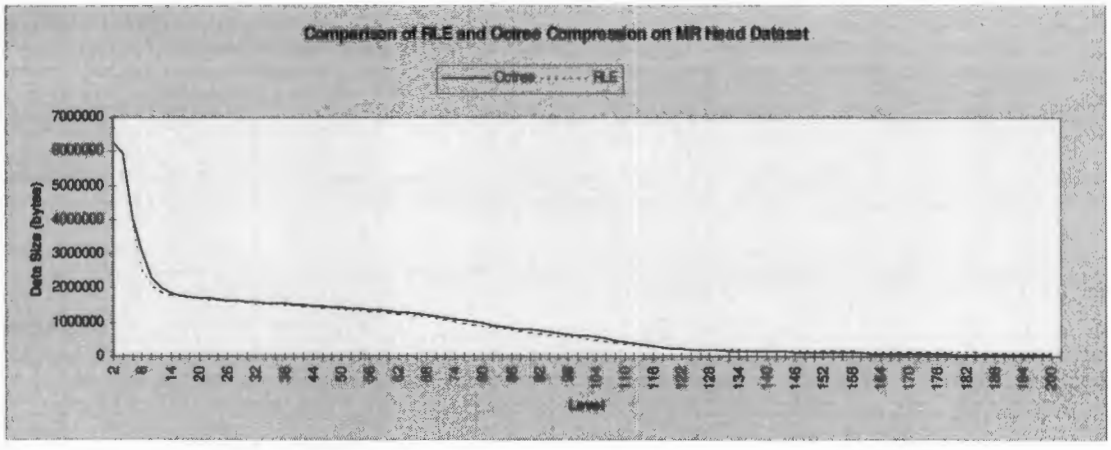


Figure 5.11 - Octree vs. RLE compression levels for the MR Head dataset.

In order to compare these two compression methods more accurately the following table presents the average differences in final volume size over each level, for the four datasets. The differences are made such that a negative value indicates better octree compression.

Dataset	CT Head	Engine	MR Head	MR Knee
Average Difference	-11 887 bytes	42 996.2 bytes	144.3 bytes	46 855.94 bytes

Considering that the original dataset size is in the order of 8Mb, these values indicate that the difference between the two methods (in terms of compression ratio) is negligible.

Unfortunately the octree compression did not improve noticeably over the RLE compression as expected. However it is clear that, even though the octree data structure contains a large amount of information for rendering the volume incrementally (i.e. Minimum and Maximum values of each sub-node, average values, etc.), the total data size compares very closely with a pure RLE compressed volume. This therefore makes the octree method more suitable for volume compression as it is giving more information about the volume and it is more flexible as a data structure.

5.6 Classification Performance

The tests presented in this section tested the performance of the classification process, where each of the voxels in the compressed data volumes are assigned surface gradients and normals.

We first present the results for the determination of the optimal cache size for use during the classification phase of a octree, and then move on to the results for the actual classification of the volumes.

Once again the classification process is performed both on the octree compressed volume as well as the RLE compressed volume, using the same parameters for comparison.

5.6.1 Determination of Optimal Cache Size

Due to the nature of octree compression, it is sometimes difficult to determine the value of a neighbouring voxel, and so the node caching algorithm was developed. However the size of the cache has to be accurately determined, as a very small cache won't be able to store the commonly accessed nodes for a particular octree branch, and a very large cache will require a lot of searching time to locate the necessary node.

Assuming that there is a fair degree of structural commonality between octree representations of different volumes (e.g. small nodes clustering next to larger nodes), it stands to reason that there is an optimal cache size range which should provide reasonable performance over most standard volume datasets.

The test thus performs the classification over a range of different octree levels, as well as different cache sizes. An octree depth of 6 is used in all cases as this was found to be the optimal depth for compression of the data, and would thus be the logical choice should the volume be transmitted to a workstation. Graphs of these results are presented in figures **C.21** to **C.24** for each of the test volumes.

In order to construct these graphs, numerous classifications were performed at each cache size, each on classifying to a different level. The average classification times were then calculated as representing the classification time for each cache size.

As expected the trends in the results are similar and it is apparent that the optimal cache size is approximately 7 in all cases. This value is a lot smaller than we expected which hints at two interesting factors:

- The problem of caching a neighbouring node is obviously a non-trivial problem as the nodes in the octree are not accessed in a neat sequential order and an LRU cache therefore struggles to maintain a genuine list on "commonly used" voxels.
- Better cache addition and searching algorithms need to be developed.

5.6.2 Performance Comparison

The classification times for the octree and RLE compressed volumes are presented below. For the classification of the octree volumes, the optimal cache size (as found above) is used. The classifications are performed at each *lower-threshold* level by both the RLE classification algorithm and the octree classification algorithm.

The summed-area table algorithm which the Min-Max octree method, in the original Shear-Warp Factorisation algorithm, makes use of is not used in either the RLE compressed volume or the octree compressed volume.

The expected performance of the octree classification stage was around 3 times slower than that of RLE classification, based on the complexity of locating neighbours in the voxel cache.

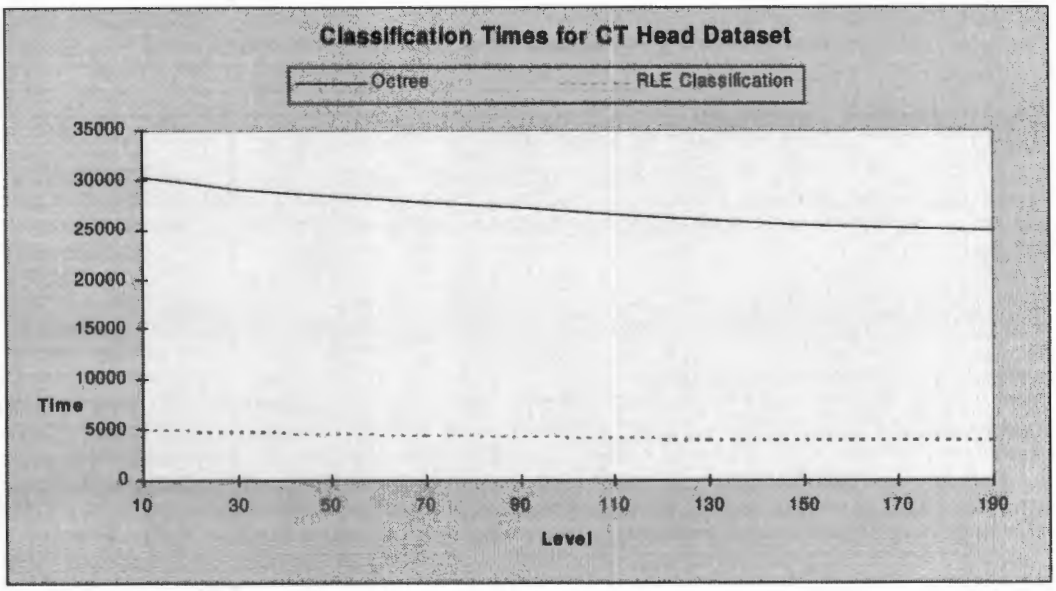


Figure 5.12 - Octree vs. RLE classification times for the CT Head dataset.

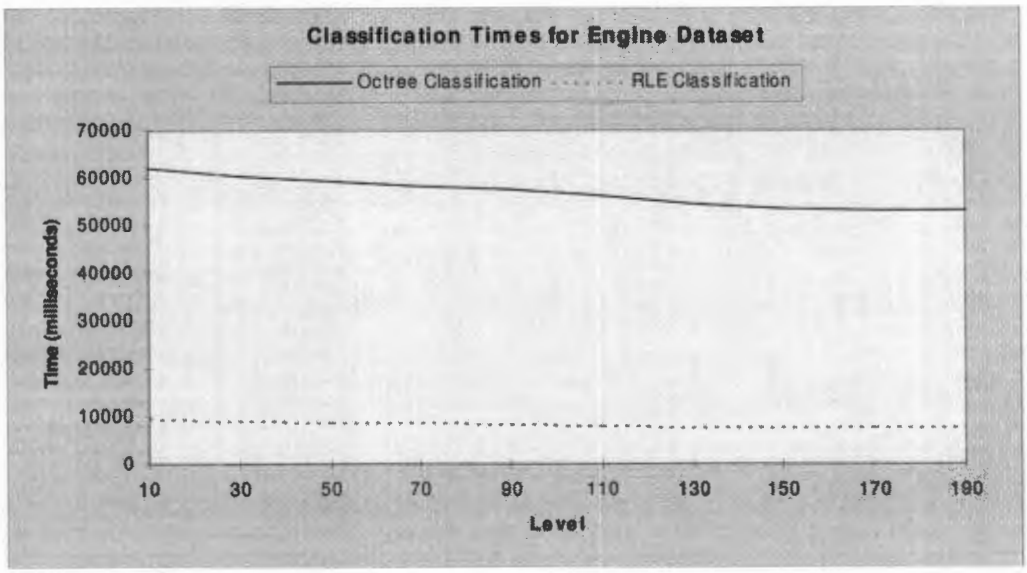


Figure 5.13 - Octree vs. RLE classification times for the Engine dataset.

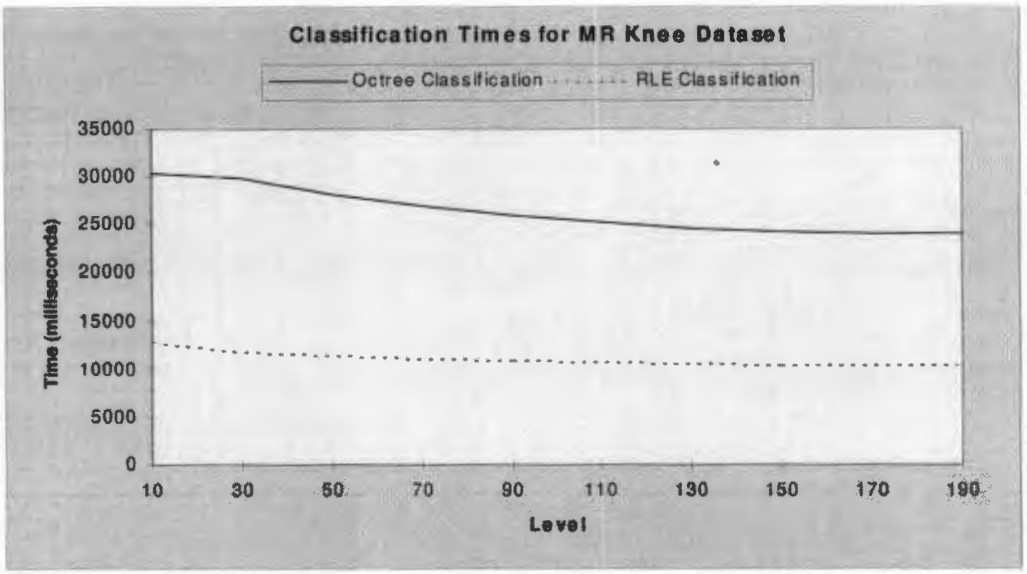


Figure 5.14 - Octree vs. RLE classification times for the MR Knee dataset.

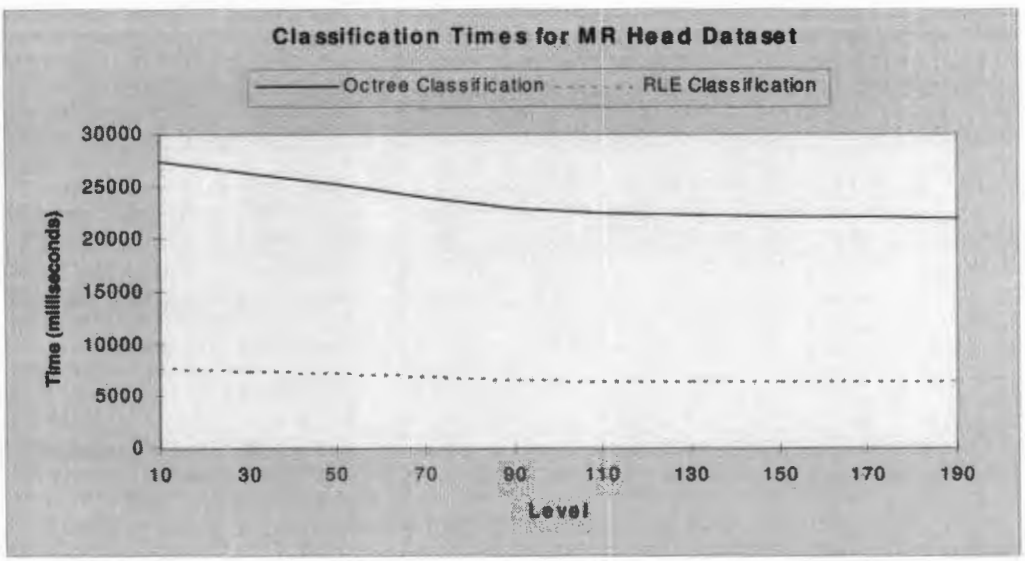


Figure 5.15 - Octree vs. RLE classification times for the MR Head dataset.

From these graphs one can see that the performance of the RLE based classification algorithm is normally between 3 and 4 times faster than the octree algorithm. This large performance loss when using the octree method is primarily due to the difficulty in classifying voxels when their neighbouring voxels lie in different octree nodes. When a neighbouring voxel is in another node the octree cache has to be used and an octree traversal can occur if it is not in the cache.

Due to the fact that the octree scheme allows for incremental classification, the data may be classified as it is received by a workstation. Thus if the total transmission time is in the order of 60 seconds or

greater (approximately the worst-case classification time from above), then the classification time will be completely amortised into the transmission time.

This advantage will, however, not be gained when the volume is re-classified later. A scheme for improving this could be to directionally thread the octree such that each node has a reference to its neighbouring node. An octree traversal will then never be necessary. This will be followed up in future research.

5.7 Rendering Performance

In testing the rendering performance of the rendering algorithms, there are a number of parameters which may be varied such as:

1. Isosurface level
2. Orientation
3. Whether a node is being rendered incrementally or not.
4. The projection matrix is perspective or parallel.

Test sequences which vary sets of these parameters were set up for both the parallel and perspective rendering algorithms, and the parameters were varied as follows:

- In each case 10 pre-classified volumes were generated at isosurface levels: 10 and 20 through to 180 in steps of 20. This parameter has possibly the most impact on the algorithms as it predicts how much of the volume may be omitted from rendering.
- The volume was rotated incrementally by: (0,15,15), (90,0,0), (90,0,0), (90,0,0) , (90,0,0), (0,90,0), (90,0,0), (90,0,0), (90,0,0), (90,0,0), (0, 0, 45), (0, 0, 45), (0, 0, 45), (0, 0, 45). (Where each set of co-ordinates represents a rotation on each axis.) These rotations ensured that all shearing sequences and front-to-back reversals were performed.
- For the octree volumes the volumes were rendered with all the data available, and then with only the tree available and no raw data. (This represents the worst possible case of the incremental rendering as every node in the volume is being approximated.)
- The octree volumes were rendered with both parallel and perspective projections matrices, however the RLE volumes were rendered only with parallel projections. (This was due to the unavailability of source code for the implementation of the traditional RLE Shear-Warp algorithm using perspective.)

The results presented in the following sections will be split between parallel and perspective rendering, and then in each section comparisons will be made between full octree rendering and partial octree rendering, as well as between full octree rendering and full RLE rendering.

5.7.1 Parallel Projection

5.7.1.1 Full Rendering

During each parallel rendering the following measurable operations are being performed:

1. Preparation of the volume, factorisation of the viewing matrix, and computation of the shading tables.
2. Rendering of the data.
3. Warping of the image.

For octree rendering, the duration of the first and third operations above was found to be constant and were as follows: (all times are in milliseconds)

	Full Render Preparation	**Full Render Warping**	**Partial Render Preparation**	**Partial Render Warping**
CT Head	1437	97	1437	97
Engine	1436	98	1441	100
MR Knee	950	87	949	88
MR Head	950	78	949	79

The duration of the same operations was also constant for RLE rendering and were: (all times are in milliseconds)

	Full Render Preparation	**Full Render Warping**
CT Head	1387	90
Engine	1410	91
MR Knee	1026	66
MR Head	940	61

The first test sequence, for which the results are presented below, is for the rendering times of performing a full rendering of the volume. The octree algorithms performance was expected to be, on the average, equivalent to RLE rendering. For lower threshold values there is less "empty space" regions of the volume and thus the octree won't represent these regions very efficiently. Therefore it

was expected that the octree algorithm should be slower at lower thresholds and then speed up as it uses higher thresholds.

Figures **C.23** to **C.26** depict the rendering times of the parallel octree algorithm. Each graph shows the average, maximum, and minimum rendering times for each of the isosurface levels configured. The average rendering time for each level comes from averaging the rendering times for each of the orientations at a particular level. The maximum and minimum values are then the best and worst rendering times resulting from particular orientations.

As expected it is clear that the octree algorithm responds well to increasing the isosurface level, as the rendering times drop steadily as the isosurface level increases. This is to be expected as the octree is causing increasingly larger areas of the original volume to be skipped out completely during rendering. Also apparent is the fact that the minimum and maximum rendering times only vary by at most 1½ seconds. When testing the RLE based algorithm a similar best-worst case difference was found, however this is only if the translated copies of the volume are calculated before any manipulation of the volume occurs. If the translation was not performed at the outset then the difference would be in the order of 8 seconds (the time taken to perform the translation in one direction plus the existing difference). The worst case rendering times were however slower than expected (for lower thresholds) which indicates that perhaps a mixture of RLE and octree compression should be used to maximise overall efficiency.

5.7.1.2 Comparison of Full and Partial Rendering

The next test script measured the performance of the octree renderer when the volume is represented only by the octree. A comparison is shown, (in Figure 5.16 to Figure 5.19), between the octree rendering times of a volume using all the data (i.e. full render) and the octree rendering times of a volume using only approximate data (i.e. partial render.) The objective here is to demonstrate that the incremental rendering times are not noticeably different from the normal rendering times.

As mentioned above the partial rendering is achieved by approximating all the leaf-nodes in the octree, which represents the worst possible case (in terms of performance) as the highest number of nodes are being approximated.

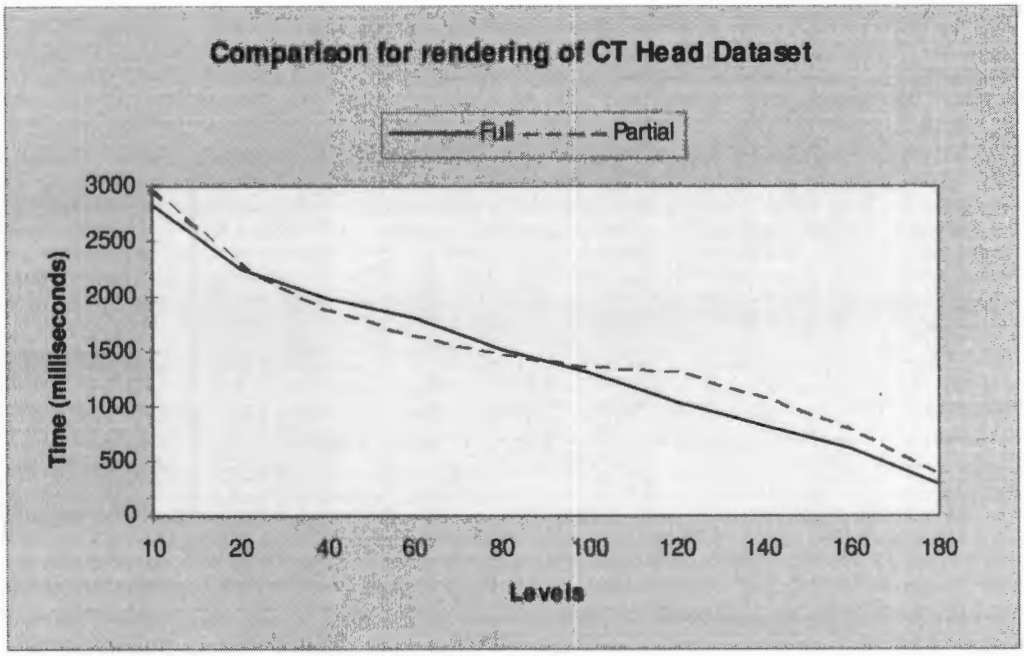


Figure 5.16 - Comparison of full vs. partial octree rendering (parallel) using the CT Head dataset.

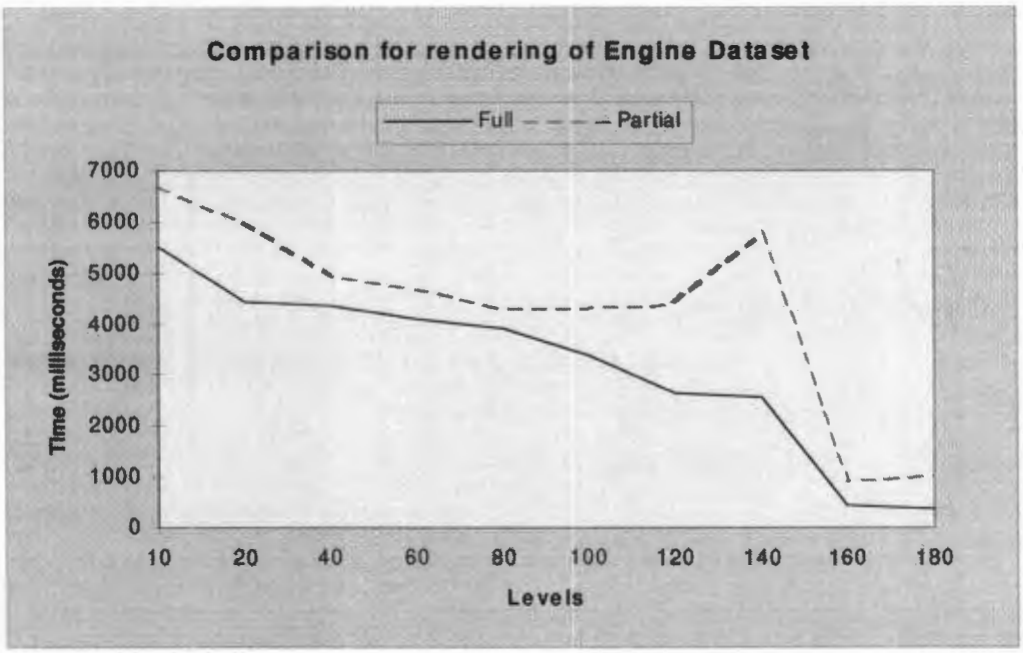


Figure 5.17 - Comparison of full vs. partial octree rendering (parallel) using the Engine dataset.

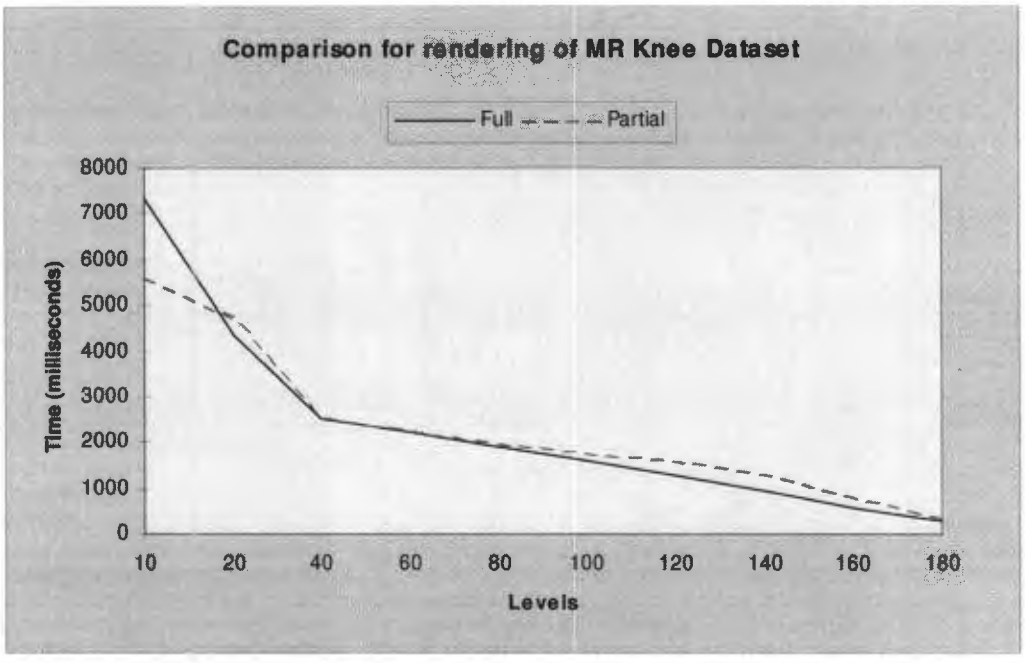


Figure 5.18 - Comparison of full vs. partial octree rendering (parallel) using the MR Knee dataset.

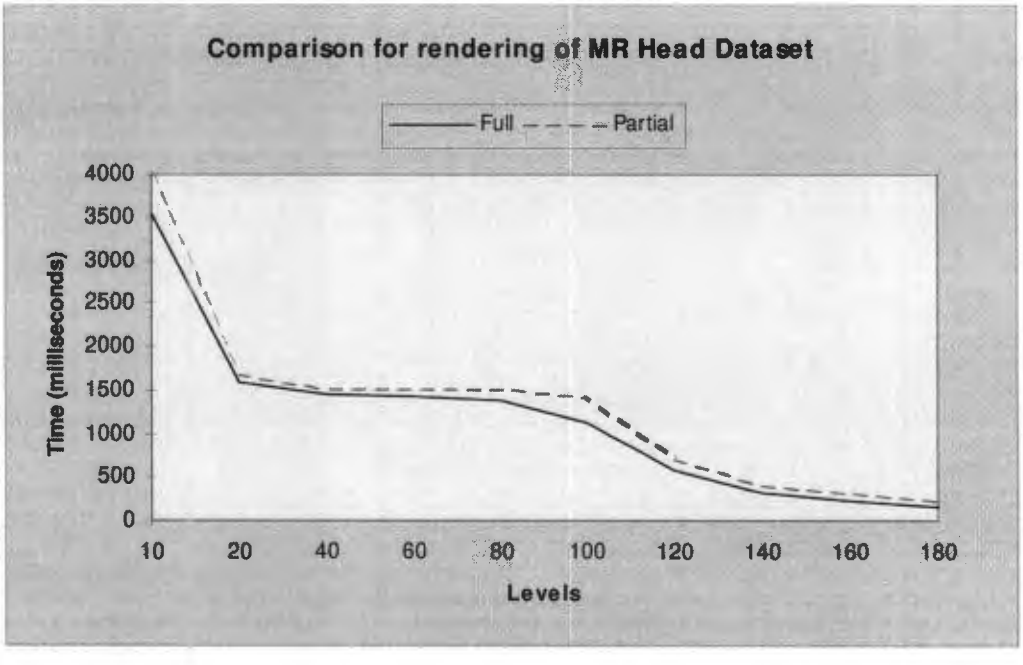


Figure 5.19 - Comparison of full vs. partial octree rendering (parallel) using the MR Head dataset.

These results are very promising and actually exceed all expectations of the renderer's performance with partial data. With the exception of the *Engine* dataset the partial rendering time is very close to the full rendering time. In some cases (normally at lower isosurface levels) the partial rendering time actually becomes less than the full rendering time. These results are very promising as they show that the rendering of a partial volume during the transmission of volume data is not going to take noticeably longer than normal rendering of the full dataset.

The peak at isosurface level 140 in the *Engine* volume's partial rendering time was not predicted however. It is probably due to the sudden generation of a large number of small leaf-nodes in the octree brought about by the isosurface level (there are small high-density components within the engine which are normally not visible). New approximation calculations thus have to be started for lots of small voxels causing a longer overall rendering time. Fortunately the rendering time is only longer by approximately 2½ seconds.

With these partial rendering times being (on average) about 3 seconds, and considering that the transferral of a volume dataset over an average Internet connection could take about 9 minutes, a more advanced approximation algorithm (tricubic interpolation, say) could be used without adversely affecting the response times of the system.

5.7.1.3 Comparison of Octree and RLE methods

The final parallel rendering test script measured the performance of the RLE rendering algorithm at the same orientations and isosurface levels as those used for the octree rendering. Figure 5.20 to Figure 5.23 depict the results of these tests and compare the results with those obtained using the octree method. These results should support the hypothesis that the octree rendering times are very close to those of RLE rendering times. The performance of the octree method was expected to be lower than that of the RLE method due to the problem with representing data at low thresholds (see §5.7.1.1), while the RLE method was predicted to be slower at higher thresholds as it is not very good at representing volumes with large areas of "empty space".

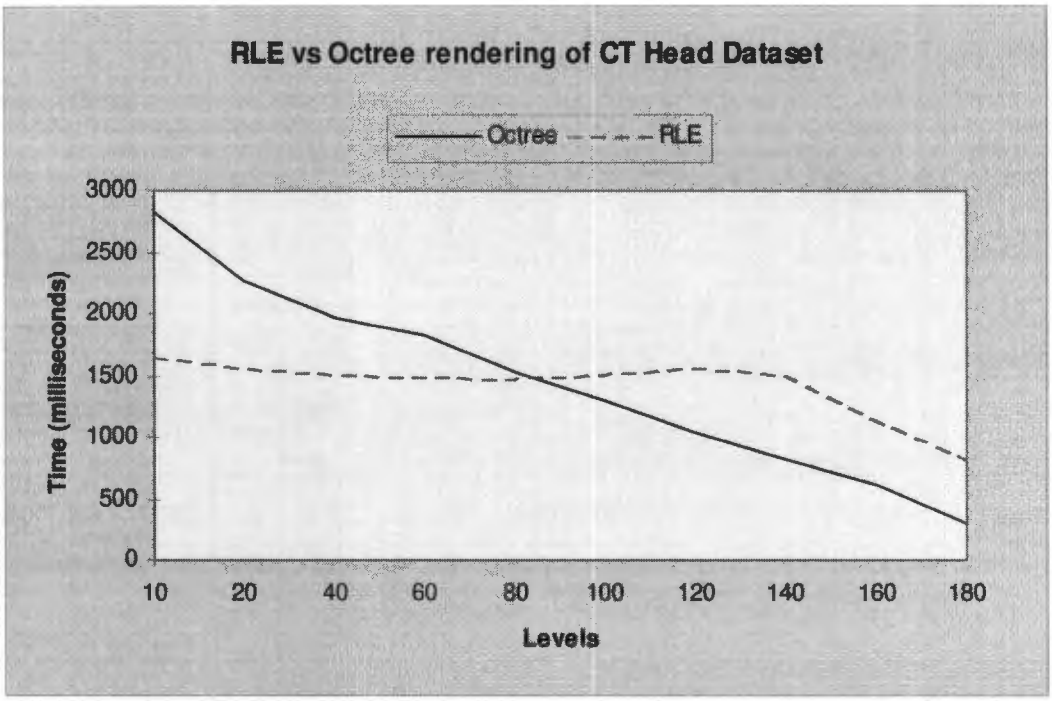


Figure 5.20 - RLE vs. Octree parallel rendering performance for the CT Head dataset.

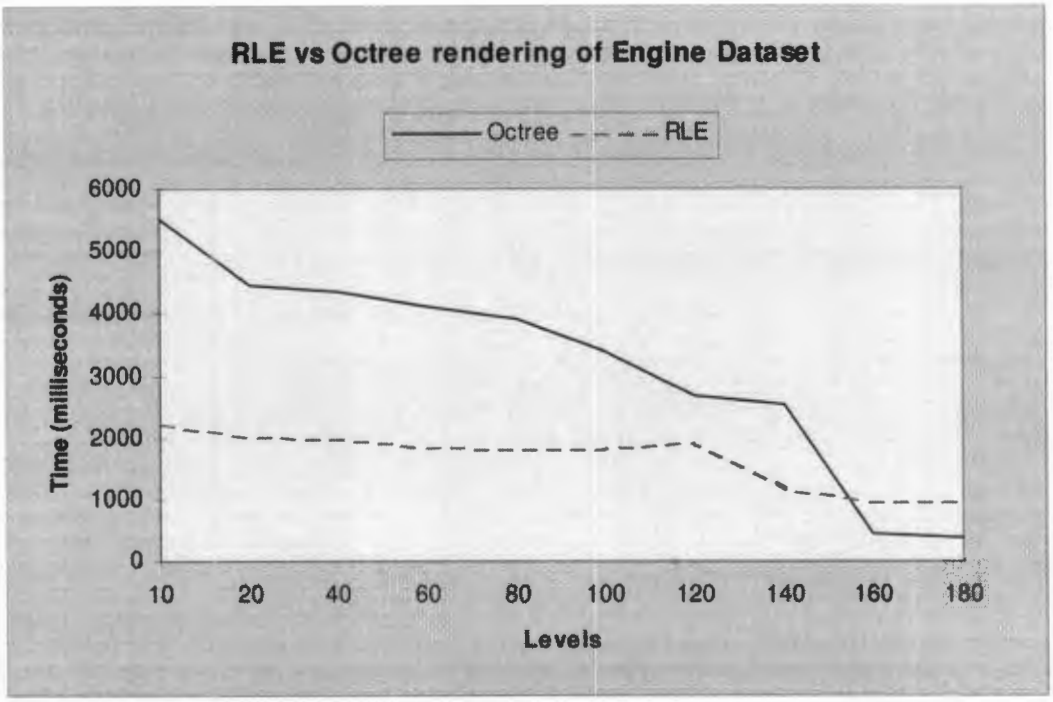


Figure 5.21 - RLE vs. Octree parallel rendering performance for the Engine dataset.

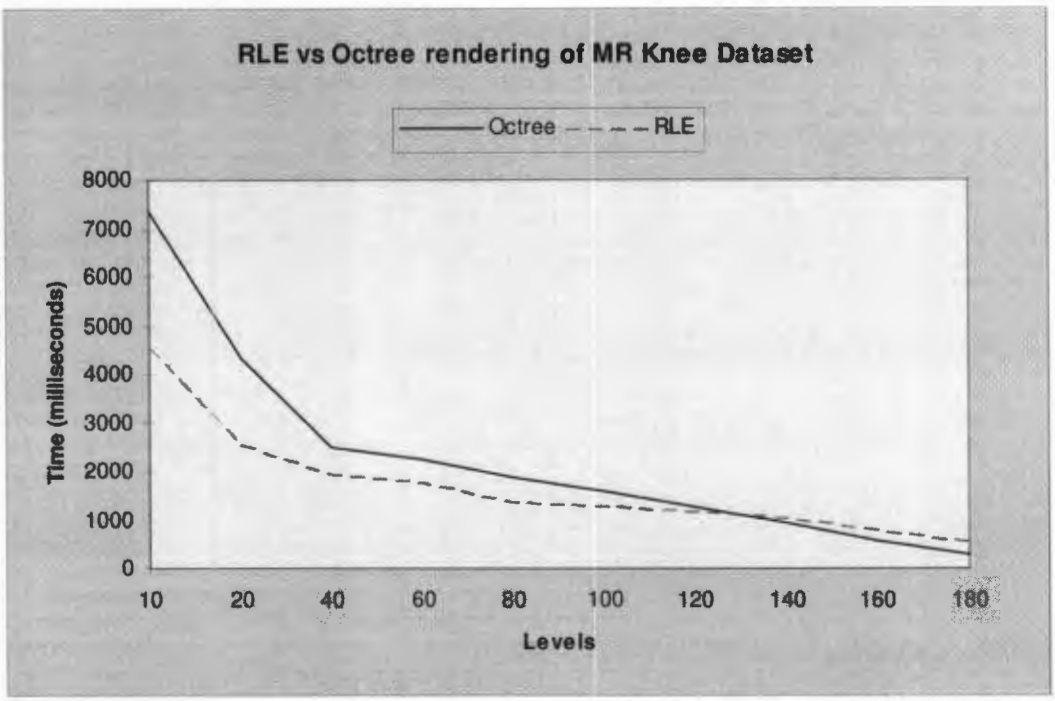


Figure 5.22 - RLE vs. Octree parallel rendering performance for the MR Knee dataset.

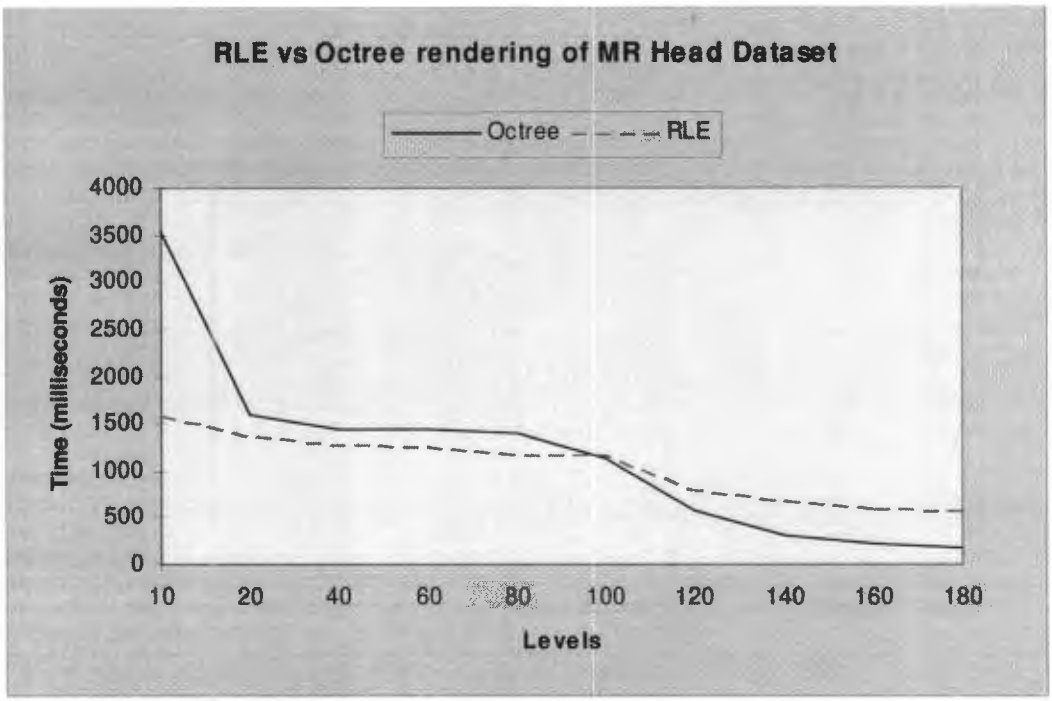


Figure 5.23 - RLE vs. Octree parallel rendering performance for the MR Head dataset.

The following trends can be seen in the above graphs:

- Octree rendering is generally better than RLE rendering for higher isosurface levels. This was expected as larger regions of the volume are being completely omitted from the rendering process, while the RLE algorithm is still stepping through them albeit slightly quicker than for lower levels.
- RLE rendering is generally better than octree rendering for lower isosurface levels. This is probably due to the greater complexity of the octree and the large number of leaf-nodes containing data which results in a lot of tree traversal. This then implies a large function-call overhead due to the increased amount of recursion. The RLE rendering method does not generate any more function calls at a low isosurface level than at a higher level, so it is more efficient in the low ranges.

The best and worst case speed improvements are given below:

	Octree faster than RLE	**Octree slower than RLE**
CT Head	64%	72%
Engine	59%	147%
MR Knee	51%	68%
MR Head	71%	121%

The octree rendering speeds were slightly disappointing as the speed of octree rendering was predicted to be almost identical to RLE rendering on the average. As mentioned above, there is a fairly large function call overhead with the octree method, so new octree traversal methods should be investigated which do not require recursion.

It should be noted however that in using the RLE rendering algorithm, it is necessary to build two transposed versions of the volume. This transposing process is fairly time consuming, and the times for this transposing process are given in the table below, for each of the isosurface levels. (The times are given in milliseconds.)

	CT Head	**Engine**	**MR Knee**	**MR Head**
10	5512	6111	7328	4369
20	5346	5922	5894	4148
40	5033	5471	5178	4227
60	5438	5422	4829	3907
80	5242	5539	4483	3386
100	5104	5692	4382	3526
120	4882	5511	4142	3254
140	4803	4979	4309	3239
160	4941	4936	3893	3218
180	4797	4917	3974	3239

Thus it is necessary to delay between 3 and 7 seconds before rendering the RLE volume at a different isosurface level.

Also implied by this volume transposing process is the fact that there are now three copies of the same volume being kept in primary storage, while with the octree method there is only one copy. A comparison of run-time memory usage is presented in the tables below for each of the volumes at an isosurface level of 20and then at an isosurface value of 160. (All values are given in bytes.) The octree method was expected to be about 50% more memory efficient than the RLE method.

(Level = 20)	RLE algorithm memory usage	Octree algorithm memory usage
CT Head	7461252	3563576
Engine	18711708	7189112
MR Knee	33570648	15966684
MR Head	20458320	7576452

(Level = 160)	RLE algorithm memory usage	Octree algorithm memory usage
CT Head	2110356	1058644
Engine	2436144	805352
MR Knee	2797740	1788604
MR Head	2031540	713980

These results are extremely promising as the results above show that the parallel octree rendering algorithm is *in excess* of 50% more memory efficient than its RLE counterpart. This supports our earlier wishes to implement an algorithm which is ideal for implementation on average workstations for the purposes of incremental volume rendering.

5.7.2 Perspective Projection

The parallel projection algorithms have now been empirically proven to be successful in terms of their performance and memory usage. This was one of the primary goals of this dissertation. It was also intended that the octree algorithm provides for perspective rendering of the volume using the same data structure. The perspective octree algorithm has not been developed to the same level as the parallel algorithm as we merely wish to prove that the perspective algorithm is feasible. No explicit comparisons between parallel and perspective rendering times will be presented here for this reason. Comparisons of image quality and performance should also not be exactly compared to published results for perspective rendering. Due to memory inefficiencies in our (non-optimised) perspective rendering algorithm the size of the MRKnee test volume prohibited it from use as a test case.

5.7.2.1 Full Rendering

As with parallel projection rendering the following measurable operations are being performed:

1. Preparation of the volume, factorisation of the viewing matrix, and computation of the shading tables.
2. Rendering of the data.

3. Warping of the image.

For perspective octree rendering, the duration of the first and third operations above was found to be constant and were as follows: (all times are in milliseconds)

	Full Render Preparation	Full Render Warping	Partial Render Preparation	Partial Render Warping
CT Head	771	191	769	190
Engine	769	191	770	190
MR Head	643	149	642	148

Unfortunately no source code has been made available for perspective rendering of RLE volumes using the Shear-Warp Factorisation method. Therefore no direct comparisons with RLE rendering can be made. However Lacroute and Levoy [24] reported perspective rendering times in the order of 2.5 seconds for a 256x256x167 size medical volume.

The first test sequence (for which the results are presented in Figures **C.27** to **C.30**) is measuring the rendering times of performing a full perspective rendering of the volume.

These graphs show the average, maximum, and minimum rendering times for each of the isosurface levels configured. The average rendering time for each level comes from averaging the rendering times for each of the orientations at a particular level. The maximum and minimum values are then the best and worst rendering times resulting from particular orientations.

The performance of the perspective algorithm was not expected to be markedly different from the parallel algorithm in terms of its relative performance improvements as the isosurface threshold increases.

As with the parallel projection the octree algorithm responds well to increasing the isosurface level. The reason for this (as mentioned before) is that the octree is causing increasingly larger areas of the original volume to be skipped out completely during rendering.

The peaks which occur at a higher isosurface levels in the rendering of the *Engine* and *MR Head* datasets came as a complete surprise. They must be due to the greater complexity of the compositing buffer brought about by the nature of the small internal high-density regions which become visible at these higher levels. Due to the scaling of the voxel slices the compositing operations can be a lot more complex, and depending on the opacity settings in the volume more compositing operations might be required. Thus sudden peaks could occur at certain levels. When an efficient perspective octree rendering algorithm is developed this aspect should definitely be considered.

5.7.2.2 Comparison of Full and Partial Rendering

The next test script measured the performance of the octree renderer when the volume is represented only by the octree. A comparison is shown, (in Figure 5.24 to Figure 5.26), between the octree

rendering times of a volume using all the data (i.e. full render) and the octree rendering times of a volume using only approximate data (i.e. partial render.) The objective here is to demonstrate that the incremental rendering times are not noticeably different from the normal rendering times.

As mentioned before the partial rendering is achieved by approximating all the leaf-nodes in the octree, which represents the worst possible case (in terms of performance) as the highest number of nodes are being approximated. The results were expected to be very similar to those obtained for parallel rendering, in that the performance differences between the two are negligible.

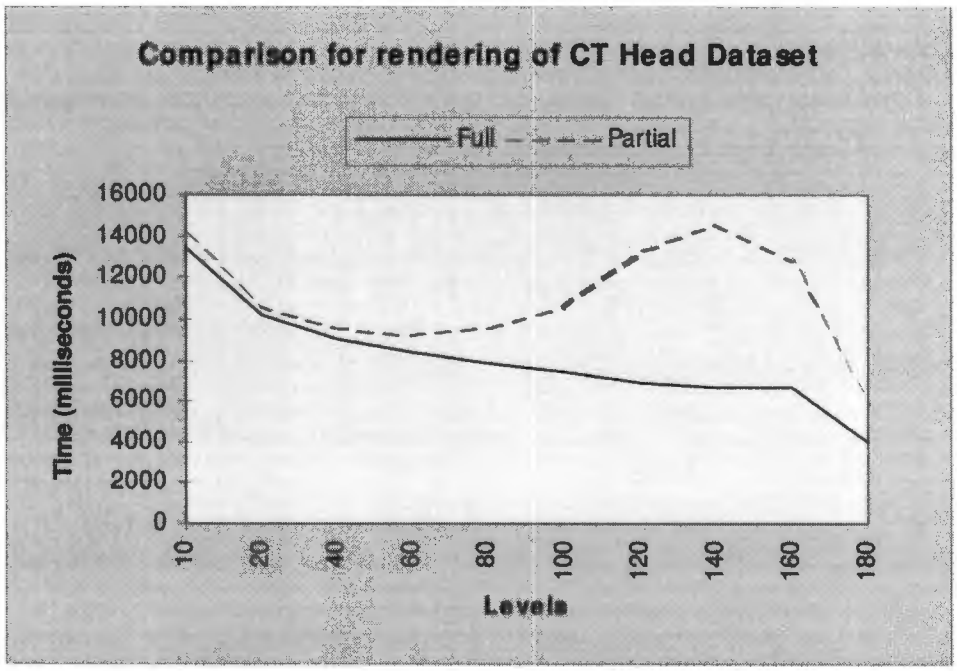


Figure 5.24 - Comparison of full and partial rendering (perspective) of the CT Head dataset.

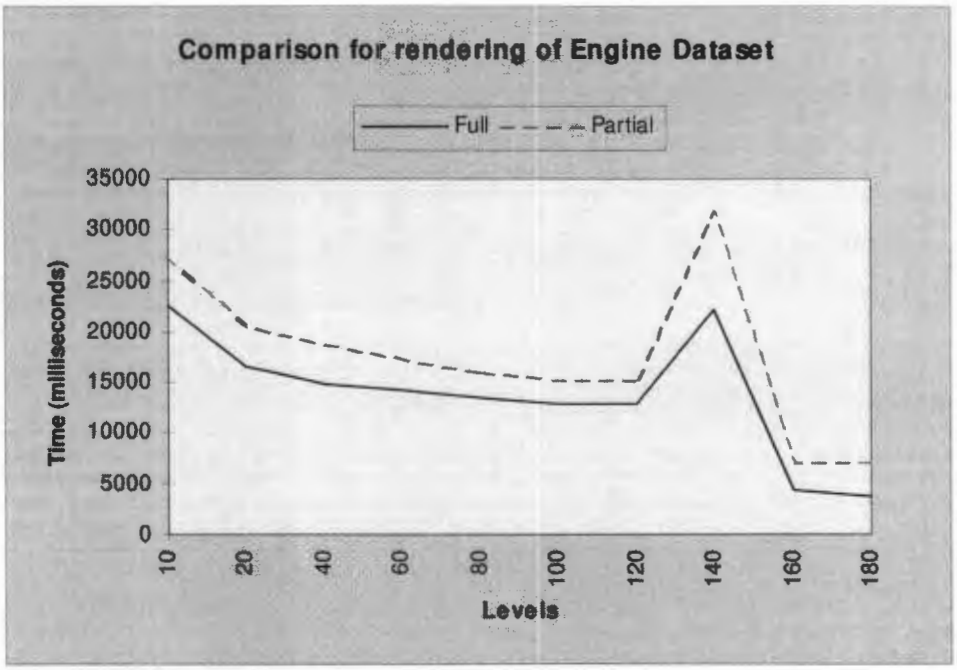


Figure 5.25 - Comparison of full and partial rendering (perspective) of the Engine dataset.

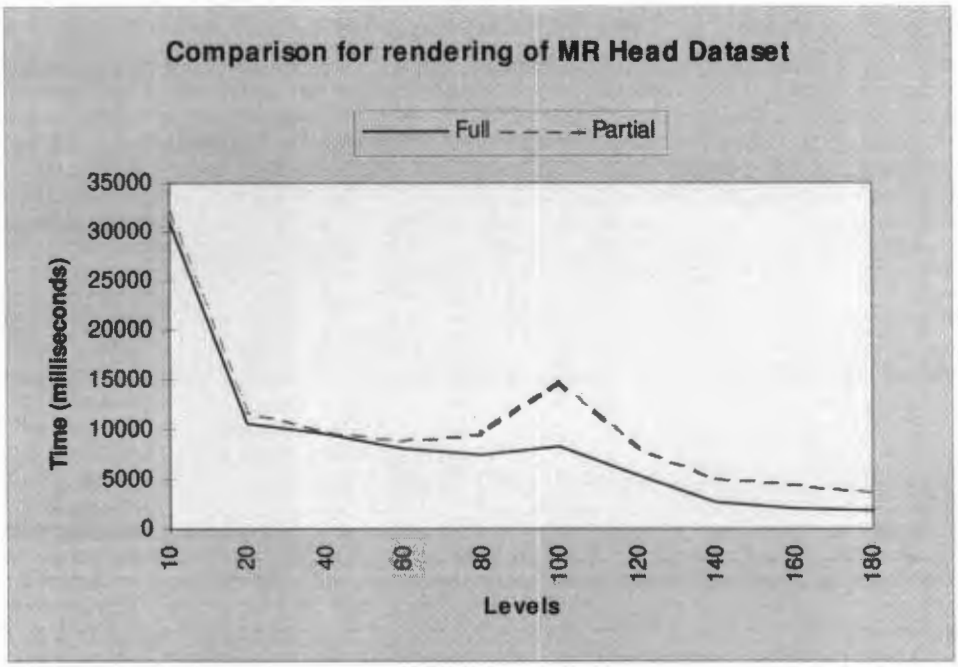


Figure 5.26 - Comparison of full and partial rendering (perspective) of the MR Head dataset.

With the exception of the *CT Head* dataset the partial rendering time is very close to the full rendering time. These results are very promising as they show that the rendering of a partial volume during the

transmission of volume data is not going to take noticeably longer than normal rendering of the full dataset.

The peak at isosurface level 140 in the *CT Head* volume's partial rendering time was a disappointing results but is probably due to the sudden generation of a large number of small leaf-nodes in the octree brought about by the isosurface level. New approximation calculations thus have to be started for lots of small voxels causing a longer overall rendering time. This combined with the greater complexity of the compositing process can cause the rendering times to suddenly peak at a particular isosurface level. Our implementation of the compositing buffer is not very efficient and a better implementation would probably increase overall performance dramatically.

The perspective rendering times presented here are between 10 and 15 times slower than their parallel counter parts. This massive drop in performance is primarily due to the insufficiency of the compositing buffer in handling scaled slices. All the slices in the perspective octree rendering algorithm are scaled by an amount greater or equal to 1. Therefore more pixels are composited into the buffer, requiring a very efficient compositing method. Future research in this area will consist of implementing the compositing buffer using quad-trees instead of run-length encoding. This will more accurately match the object order traversal and should therefore offer a speed improvement.

The same results concerning the translation of the volume in the RLE case apply here. As mentioned earlier in this section, no direct comparisons are available with perspective RLE rendering. Considering the average rendering times of perspective volumes was reported to be roughly 2.5 seconds, these results show a performance drop of between 5 and 8 times. However, we have shown that perspective rendering is also covered by the octree algorithm.

We now move on to visually comparing the rendering results of the various methods mentioned above.

5.8 Images and Animations

For each of the test volumes, two standard view points were chosen which best reflect the contents of those volumes. A variety of renderings of these volumes using both the octree method and the RLE method are presented in *Appendix D*. The renderings are presented in four sections:

- *Standard Parallel Renderings* - A comparison of normal opaque surface renderings of the full data set using both the parallel octree algorithm and the parallel RLE algorithm.
- *Standard Perspective Renderings* - Presents renderings of three volumes from a primary view point using the perspective octree algorithm.
- *Using Translucency* - A comparison of semi-opaque and opaque renderings of the engine dataset using both the octree algorithm and the RLE algorithm.
- *Partial Renderings* - Partial octree renderings of the volume data using a variety of approximation levels as well as parallel and perspective rendering.

These images show that the image quality of the octree rendered volumes suffers slightly in that the aliasing is a lot more noticeable than for the RLE rendered images. This is due to the zero-order interpolation being performed in all directions as opposed to the bilinear interpolation performed by the RLE algorithm. However for semi-opaque objects and regular geometry objects the aliasing effects are barely noticeable. Future research should therefore go into introducing filtering to the octree algorithm. Fortunately due to the three-dimensional symmetry of the octree data structure any filtering process which is performed can be performed with equal ease in all directions. Thus should linear filtering be introduced to the octree algorithm (say through the use of overlapping octree nodes) then trilinear interpolation may be performed through the volume, making it superior to the RLE algorithm.

Another noticable effect which occurs in images of the *CTHead* data set, when it is approximated, is holes in the surfaces. This is a side-effect of the trilinear interpolation method for approximating missing regions using mainly the data in the 8 corners of a voxel region. In the case of the *CTHead* dataset the skull area is very dense but very thin so it often arises that the eight corner values end up lying on either side of the skull thus averaging out to a very low density sub-volume. Hence the holes.

Numerous animations were developed to further validate the effectiveness of the octree volume renderer. Many of the aliasing artifacts are more noticeable under animation. These animations are available over the World-Wide-Web at *http://www.cs.uct.ac.za/~mikeh*.

5.9 Conclusion

Numerous hypotheses were presented in this chapter concerning the performance of the hierarchical volume representation and rendering algorithms. A set of tests were then devised which supported these hypotheses.

Primarily the ability of the octree algorithm to effectively compress the volume data (to a level comparable to RLE compression) as well as the ability to perform approximate renderings at no extra cost, was demonstrated. Another very important result was that the runtime memory usage during rendering is reduced by more than 50% when using the octree algorithm instead of the RLE algorithm.

Other results showed that leaf-node compression is advisable during the transmission of the volume, and that the decompression overhead at the client end is negligible. The parallel rendering performance using the octree algorithm was found to be close (although on the average slightly slower due to function call overhead) to that obtained using the RLE algorithm (in some cases slower and in other cases faster). Also perspective rendering using octrees was shown to be possible.

Results concerning the classification times of the volumes (when using octree representations) were not very promising. The classification times of the volume were found to be in the order of 3 to 6 times slower than the RLE algorithm, despite the use of an octree node cache. Fortunately on initial transmission the volume may be incrementally classified as it arrives, thus amortising the classification time over the transmission time. This is not possible with the RLE representation method, however on later re-classifications of the volume the RLE method is faster than the octree method.

In conclusion the results presented in this chapter completely validate the octree algorithm's ability to perform efficient and incremental volume rendering on a wide range of lower-end workstations. This is achieved through a highly efficient rendering algorithm, low runtime memory usage, and incremental rendering.

Chapter 6

Conclusion

6.1 Overview

The research presented in this dissertation was primarily aimed at exploring new methods for volume rendering which would work efficiently on average desktop workstations. Another aspect of the research was to ensure that this method could be distributed over a conventional network (such as the Internet) in such a way that network transmissions are minimised and rapid user feedback is maintained. The algorithms presented in this dissertation are also presented in [57].

The central theme of the dissertation was the use of the octree data structure for representing the original volume dataset in a compressed form. This data structure was selected for its hierarchical nature as well as its three dimensional nature. By selecting an isosurface threshold level for the volume much of the data in the original volume may be filtered away. However the resulting "shape" still has to be represented as efficiently as possible for rapid transmission over a network or for storage in primary memory. Our octree data structure represents the filtered data exponentially better as the depth of the octree increases. However with an increased depth in the octree comes an increase in the size of ancillary data structures. A compromise was found in using an octree of moderate depth while the data at leaf nodes is further compressed using a run-length-encoding scheme. Unfortunately this leaf compression makes the data too complex to reference during rendering so this leaf compression is only used during network transmission.

By inserting extra information into the nodes of the octree the hierarchical nature of the octree allows regions of the volume to be approximated at varying levels of accuracy. A novel result of this ability is that incremental transmission and incremental rendering of the data is supported. In other words, as the octree volume data arrives over the network an approximated volume is rendered. Then as more data arrives over the network the volume is constantly re-rendered, each time improving in its approximation of the original volume. If data were being transmitted over a relatively slow network link then user-feedback is still being maintained at the expense of a more approximated volume.

As the rendering of the octree had to occur sufficiently fast for numerous renderings to occur during the transmission time of a volume as well as perform efficiently on average workstations, a highly efficient rendering algorithm was required. The Shear-Warp Factorisation method was used for this purpose. However the initial Shear-Warp algorithm made use of Run-Length-Encoding (RLE) data structures for representing volumes. Our algorithm modifies the basic Shear-Warp algorithm to work with octree data structures instead. Many aspects of the original algorithm are centred around the unidirectional and linear nature of the RLE data structures so the modifications were quite extensive and numerous advantages over the original data structures were found.

One of the primary problems with the use of the original RLE data structure was the necessity of maintaining three transposed copies of the volume. This was due to the unidirectional nature of the data structure. Our octree method is however symmetrical in three dimensions and thus removes this limitation completely.

Another feature which the octree data structure has is its separability from the raw volume data. In the case of run length encoded data, the data comprises one large array of raw data with encoded runs inside it. However our octree data structure is a completely separate entity with references into an array of raw data. This property allows the maintenance of numerous other octree structures which contain other information or which filter the data differently, which can accelerate rendering and certain other processes.

As mentioned above one of the main advantages of our octree data structure is that it can contain approximation information and can therefore provide for approximate renderings when not all of the data is present. The algorithms which we developed make use of trilinear interpolation during the rendering process to approximate entire three dimensional regions of the volume for which there is no raw data available.

The main problem experienced with moving to the octree data structure was the performance of the volume classification phase which has to occur just before rendering. This phase requires the computation of values based on neighbouring sets of voxels. When attempting to compute these values for voxels lying on the edge of an octree node it becomes very complex to determine the values of neighbouring voxels due to constant octree traversals. To alleviate this problem an octree node caching algorithm was developed.

Algorithms for rendering the octree represented volume using both parallel and perspective Shear-Warp rendering were developed. The parallel rendering algorithm was fairly similar to the original RLE based algorithm except that the order of rendering voxels becomes more complex. This is due to the fact that the data in the volume is no longer stored in a convenient scan-line order, but rather it a number of hierarchically placed nodes. An octree traversal algorithm was developed which rendered nodes in the octree in such a way that occlusion of voxels in the compositing buffer were still correct. The only problem experienced during the development of the parallel rendering algorithm was the need to omit the bilinear filtering which was performed in the original RLE based algorithm. This is also

due to the same reason that classification is slow. (i.e. difficulty in determining the values of neighbouring voxels.)

The perspective rendering algorithm which we developed was a lot more complex than the original Shear-Warp algorithm due to the need to average sets of voxels together as the distance from the viewing plane increases. Once again the scan-line ordering of the original RLE data made this a reasonably simple task However the averaging of regions of voxels together in the octree model becomes very complex due to the difficulty of locating neighbouring voxels at the edge of an octree node. Also the octree traversal algorithm used for parallel rendering was shown to break down under perspective projection. New algorithms were thus developed for perspective rendering which: (a) moved the region averaging process into the final warping of the compositing buffer, and (b) traversed the octree in an adaptable fashion which still insured correct occlusion.

6.2 Results

An extensive set of tests were devised to verify the use of the algorithms (see Chapter.5) mentioned above and to show that noticeable improvements are possible through use of the octree data structure.

The ability of the octree to compress the data was extensively tested. The performance of the compression process was found to be under 10 seconds and considering that this process generally only has to occur once when viewing a volume it is very acceptable. It was also shown that the leaf node compression technique is very successful and that if it is used during transmission then the overall sacrifice in performance is no longer than a few seconds. The compression ability of our octree algorithm when compared with the RLE algorithm was found to result in data structures of roughly the same size. This result was initially thought to be disappointing but when considering that the octree data structure contains a lot more information about the structure of the volume as well as the approximation information this result is excellent.

Tests were performed on the octree classification algorithm to determine the size of the cache to be used, and an optimal cache size of 7 was selected. This value was a lot lower than expected and indicates that the caching algorithm is not making a sufficient improvement to the performance. When measuring the classification times of octree data volumes against RLE data volumes it was found that the RLE algorithms performed three to four times faster than their octree counterparts. This was a very disappointing result (see section §6.3), however when using these algorithms over a network and performing incremental rendering the classification process is also incremental and is thus amortised over the duration of the data transmission. While this is certainly an advantage, later re-classifications on the client workstation will incur the full performance penalty of octree classification.

After classification the tests moved on to measure the performance of the parallel and perspective rendering algorithms. For each algorithm the performance when rendering a full data structure was compared with the performance when rendering a volume which is being maximally approximated. The tests found (for both parallel and perspective rendering) that the performance of the approximation rendering was equivalent (and in some cases faster!) to the normal full rendering. This was an

extremely promising result and indicates that more advanced approximation algorithms can now be used without adversely affecting the overall performance.

The memory efficiency of both the parallel and perspective rendering algorithms were measured and compared with the RLE method. It was found that our octree method used *less* than half of the memory required by the RLE method. This serves to strengthen our argument that the octree algorithm makes rendering more suitable to average workstations with low primary memory capacities.

For parallel rendering the octree rendering performance was compared with RLE rendering performance over a variety of datasets and settings. It was found that for lower isosurface threshold levels the RLE algorithm performed better and for higher isosurface levels the octree algorithm performed better. This confirmed our expectations that the octree data structure would cause the rendering to be a lot more efficient when large areas of the volume are filtered out. On average the octree and RLE rendering algorithms were found to be almost equivalent in execution time. When considering that the octree algorithm also provides for incremental rendering and uses less than half the memory of the original algorithm this result is very acceptable. Thus the goal of developing an algorithm which is capable of rapidly re-rendering a volume repeatedly during its transmission has been attained.

The perspective rendering performance was disappointing, however no effort was made in the research to optimise the perspective algorithm due to its complexity. The tests of the perspective algorithm were designed merely to prove that perspective rendering was feasible with the octree method. The execution times of the perspective renderings were found to be in the order of 10 to 15 times slower than their parallel equivalents, and 5 to 8 times slower than figures published for the original Shear-Warp perspective algorithm which used RLE data structures. An unexpected drop in the performance of the perspective octree rendering algorithm was also found for certain isosurface levels, which indicated that the function call overhead of the recursive octree traversal is probably adversely affecting the result.

Sets of renderings were then executed on numerous test datasets at various orientations and isosurface levels to compare the quality of renderings between the octree and RLE methods. The omission of the bilinear filtering step in the octree algorithm was found to introduce noticeable aliasing artifacts into the images.

6.3 Future Work

The only really disappointing result in this dissertation was the inability of the octree data structure to handle situations where neighbouring voxels had to be rapidly located. This adversely affected the classification time of the volume as well as the quality of the rendered images. Future work thus needs to concentrate on solving this problem. There are two possible solutions to this problem which could be attempted. Firstly the octree could be *directionally threaded*, meaning that spatially neighbouring octree nodes have references to each other. This however incurs a fairly large overhead on the size of the octree and still requires a degree of computation to determine which voxel neighbours another.

The second approach and the most likely to succeed would be to overlap each octree node by one voxel. This then implies the storage of redundant data however it makes the location of neighbouring voxels extremely rapid. By using this approach, the classification stage could be accelerated to be at least as efficient as the RLE method. The only problem experienced which this method will not solve is the rapid averaging of regions of voxels during the perspective splatting process (although this could still be overcome by our more data intensive process of expanding the image towards the front instead of reducing it toward the back). However a novel result of this approach would be that trilinear filtering could then be used during the rendering process which should (theoretically) produce better images than those produced with a bilinear filter. Design of this algorithm should be fairly straight forward and it could be designed and validated within a month.

The transmitted volume size is still extremely large and more efficient leaf-node compression techniques are probably required. An advantage of having the octree breaking the volume up into numerous different sized nodes is that different leaf-node compression algorithms could be used for different sized nodes. Then algorithms such as DCT based compression or vector quantization could be used to compress the leaf nodes resulting in a much smaller overall data structure. During transmission, even the octree data structure could be compressed using a stream based compression technique such as Lempel-Zif compression. Development of algorithms to perform this compression would not be difficult and would only impact a very small part of the overall architecture proposed in this dissertation. Based on the published results of these other compression techniques reductions of a further 50% to 80% could be expected.

Another improvement which will serve to decrease rendering times (especially in the perspective rendering case) is the use of quadtrees in the compositing buffer. Due to the RLE-type structure of the compositing buffer it is not ideally suited to the traversal order of the octree during rendering. By modifying the compositing buffer to use a hierarchical quadtree, entire regions of the octree may be efficiently skipped from rendering when their impact on the compositing buffer is not noticeable. The complexity of developing an algorithm to do this is largely unknown but is assumed to be non-trivial and thus could take quite some time to develop.

Appendix A

Glossary

affine — A combination of scales, rotations, shears, and translations. An affine transformation is one which preserves the parallelism of lines.

aliasing — Low frequencies in a signal which are actually high frequencies. This occurs if a signal is reconstructed using a sampling rate above the Nyquist limit (equal to double the frequency of the highest frequency component in the spectrum).

bilinear filtering — Linear filtering (or neighbour averaging) in two directions.

child node — A node of the tree which forms part of a larger node. In an octree a child node represents a sub-volume of the parent node's volume.

compositing — A process by which an image is added or merged with another image.

DCT — Discrete Cosine Transform. A reduction of the Fourier Transform to one real component. Often used in image compression.

image warping — The mapping of one image to another using some form of two dimensional transformation. This transformation may be either linear or non-linear.

leaf node — A node in a tree which has no child nodes. Leaf nodes in an octree are the only nodes which directly reference volume data.

lossless compression — Compression of data such that on decompression the original data is exactly restored.

lossy compression — Compression of data such that on decompression an approximation of

	the original data is generated.
node	A point in the octree which represents a small cube of data in the volume. The size of this cube of data depends on the depth of the node within the octree.
normal vector	A vector perpendicular to a surface and with a magnitude based on the curvature of the surface. In vector calculus this is equivalent to the grad vector of a surface.
occlusion compatible	Making sure that objects which should appear behind others still appear behind, and objects which should appear in front still appear in front.
octree	An abstract tree data structure where every node has at most eight children. This may also be visualised as the recursive subdivision of a cube along each of the three axes to generate eight smaller cubes.
opacity level	The level of non-transparency of an object. A high value indicates a solid surface which cannot transmit light, and a low value indicates a transparent surface which can transmit light.
opacity transfer function	A function by which the opacity of a single voxel is determined. This is normally a function of the voxel's value and its gradient magnitude.
over operator	A method of compositing.
parent node	A tree node which has child nodes associated with it. In an octree parent nodes never directly reference raw volume data.
permutation matrix	A matrix which permutes the X, Y, and Z factors in a transformation matrix.
projection matrix	A matrix which projects points onto an image plane to create an image of the object represented by the points.
pyramid representation	The hierarchical decomposition of an image or volume into a fully branched tree.
quadtree	An abstract tree data structure where every node has at most four children. This may also be visualised as the recursive subdivision of a square along each of the two axes to generate four smaller squares.

RLE	Run-Length-Encoding. The compression of data by encoding runs of equivalent values into a single datum.
Shear-Warp Factorisation	The factorisation of a transformation matrix (which orientates and projects a solid on an image plane) into a three dimensional shearing and scaling matrix and a two dimensional image warping matrix.
spatial gradient	A 3-vector representing the rate of change of values in a volume at any given point for each orthogonal direction. This is usually calculated using finite differences.
splat	The process of compositing a region of pixels onto the image plane.
sub-volume	An axis aligned cubical region of the original full volume dataset.
trilinear interpolation	Interpolation between eight points in a three dimensional space using linear combinations of the eight points.
voxel	A three dimensional sub-volume in a volume which cannot be further subdivided and which contains one single value. (The three-dimensional analogy of a pixel.)

Appendix B

The VOX++ class library

B.1 Introduction

During the development of the octree compression and rendering algorithms presented in this dissertation it was necessary to implement and validate the various algorithms. Due to the fact that the octree algorithms take effect at various stages in the volume visualisation pipeline, a comprehensive framework was required for these algorithms to be implemented. This framework should ideally cater for all stages in the processing of volume data in an easily extensible fashion.

On examining the various forms of data which were processed or produced during the visualisation operations, it became clear that there was a very strong relationship between operations and certain types of data. This indicated that some form of object encapsulation was required to elegantly represent both the data and the operations. Generalisations of certain sets of data types were also identified indicating that some form of inheritance and perhaps polymorphism were required. It was thus decided to implement a class hierarchy of objects which implement and assist the various stages of volume visualisation.

This class library (referred to as *VOX++*) was implemented in C++ and provides a reasonably flexible and extensible direct-volume rendering framework. The class and object hierarchies are presented below, and thereafter a class reference is provided for the various objects in the class library.

B.2 Class and Object Hierarchies

The class hierarchy for the class library is depicted in Figure B.1. The main hierarchy is the DATAOBJECT derived hierarchy. The DATAOBJECT class has two derived objects, VOLUME and IMAGE. VOLUME derived objects contain all of the compression and rendering methods and the volumetric data itself, while IMAGE derived objects contain all of the two-dimensional image manipulation routines. A number of numeric linear algebra classes are provided: 3 and 4 vector objects as well as 3x3 and 4x4 matrix objects. The 4x4 matrix object (MATRIX4) has three inherited objects which are special cases of a homogenous 3x3 transformation matrix. These three matrix objects allow the manipulation of the orientation of a volume. The PARAMETER hierarchy provides

a number of very simple objects which encapsulate the various forms of parameters that can be supplied during the visualisation process. Parameter objects for: opacity table definition, object orientation, light definition, material definition, and image format are provided.

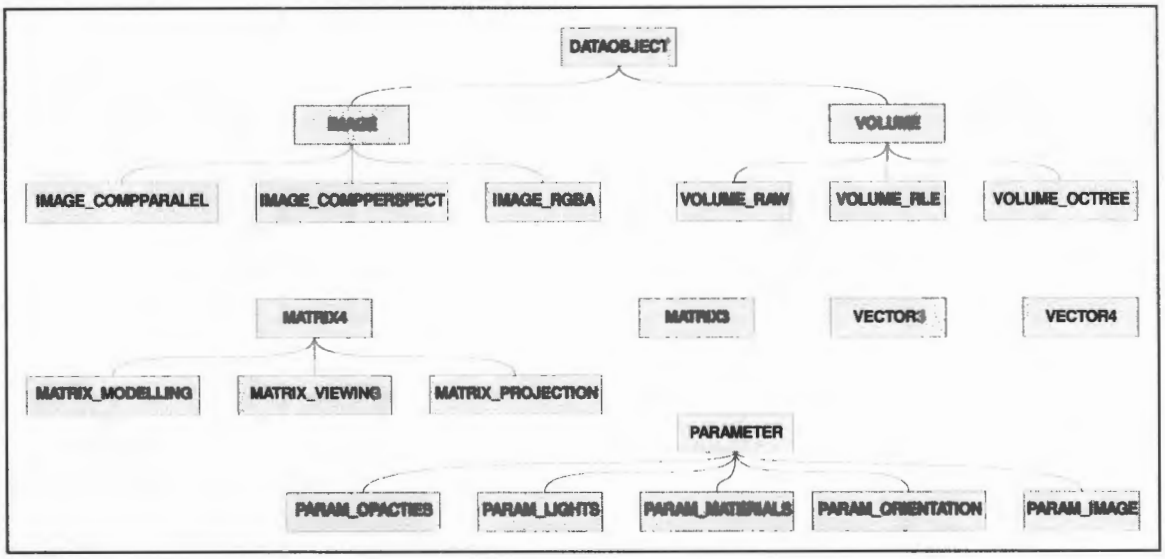


Figure B.1 - Class hierarchy for the VOX++ class library.

The object hierarchy of the class library is depicted in Figure B.2. This shows the relationships between the various objects as either:

- *Input/Output Relationships* - The source object is either passed as a parameter to the destination object, or the destination object is produced by the source object.
- *Association Relationships* - The source object is contained inside, or used temporarily within, the destination object.

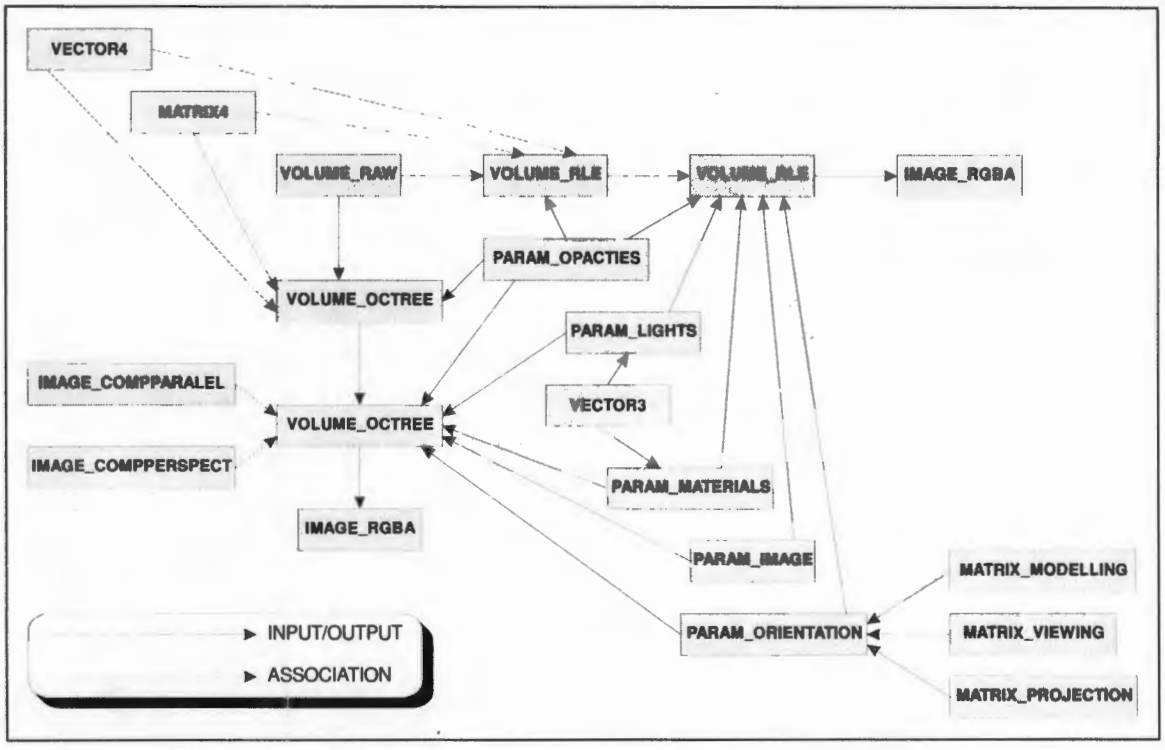


Figure B.2 - Object hierarchy for the VOX++ class library.

The standard flow of control is as follows:

1. Create a VOLUME_RAW object to contain the original raw data. This filters and optimises the original data.
2. Create either a VOLUME_OCTREE object or a VOLUME_RLE object from this VOLUME_RAW object. This creates the compressed data structure.
3. Pass a PARAM_OPACITIES object to the VOLUME_RLE or VOLUME_OCTREE object to classify the volume and produce another VOLUME_RLE or VOLUME_OCTREE object.
4. Pass instances of all PARAMETER derived objects to the new VOLUME_RLE or VOLUME_OCTREE objects to render an image. An IMAGE_RGBA object is produced which contains the two-dimensional image.
5. Save the IMAGE_RGBA object to an image file.

B.3 Class Reference

B.3.1 DATAOBJECT

Description: Abstraction of all main data objects in the system. It forms the basis of encapsulation for all objects which contain actual visualisation data and which operate on that data. Objects of this type are never directly instantiated.

Members:	DataType	An enumerated type containing the exact type of data (e.g. classified octree volume data, or RGB image data). This allows for a degree of runtime type checking.
	RefCount	A reference count for how many times this object is being referenced. This is useful in an event driven GUI where a single object may be used in a number of different windows. When the object is no longer required in one window it should not necessarily be destroyed as it might still be required in another window.
	Flag_Success	A flag which indicates that the previous operation on the data object was successful.
	Flag_Locked	A flag which indicates that the data object is locked from modification. Once again this is useful in a windowed environment where one data object may be altered through many windows.
	Flag_Allocated	A flag which indicates that the data associated with this object (e.g. the actual volume or image data) is currently allocated in primary memory.

Methods:	DATAOBJECT	Constructor. Accepts the object type as a parameter.
	OK	Returns the value of the *Flag_Success* member indicating the result of the previous operation.
	CanRead	Returns whether the object may be used in a read capacity.
	CanWrite	Returns whether the object may be used in a write capacity.
	Ref	References the object. (Increases the reference count.)
	DeRef	De-references the object. (Decreases the reference count.)
	Lock	Locks the object from write modification.
	Unlock	Unlocks the object from write modification.

B.3.2 IMAGE

Description: Abstraction of all two dimensional image objects in the system. This object provides the common memory functions and access functions for images. Objects of this type are never directly created.

Members:	XLen	The X size of the image in pixels.
	YLen	The Y size of the image in pixels.

Methods:	IMAGE	Constructor. Accepts the image type as a parameter.
	Allocate	A virtual method which allocates the necessary memory for the image data based on the X and Y size of the image.
	Access	A virtual function which returns a memory pointer to the allocated image data.
	SetSize	Sets the X and Y size of the image.
	GetXSize	Returns the X size of the image.
	GetYSize	Returns the Y size of the image.

B.3.3 IMAGE_COMPPARALLEL

Description: A 32-bit per pixel image with an alpha channel for opacity. This object is used during the slice compositing process in the parallel Shear-Warp Factorisation algorithm. The image has a secondary data structure which encodes runs of opaque or semi-transparent pixels for accelerating the compositing process.

Members:	PixelData	A pointer to a memory block containing the pixel data for the image.
	AccelList	A pointer to a memory block containing the opaque/transparent run information. This consists of an array per scanline.
	TotalPixels	The total number of pixels in the image.

	TotalCells	The maximum number of run encodings in the image.

Methods:	IMAGE_COMPPARALLEL	Constructor.
	Clear	Clears the image to all black pixels with 0 opacity.
	NewScanLine	Starts a new scanline for compositing and returns a reference to a position in the list of runs.
	AddOpaqueCell	Adds a new completely opaque pixel into the list of runs.
	IsOpaque	Validates whether the specified pixel is completely opaque or not.
	Run	Returns the number of pixels to skip or process from the specified position. (If the returned value is <0 then that number of pixels can be skipped, otherwise it is the amount that has to be processed.)

B.3.4 IMAGE_COMPPERSPECT

Description: A 32-bit per pixel image with an alpha channel for opacity. This object is used during the slice compositing process in the perspective Shear-Warp Factorisation algorithm. The image has a secondary data structure which encodes runs of opaque or semi-transparent pixels for accelerating the compositing process. This object also allows for a re-sizing factor for each of the slices.

Members:	PixelData	A pointer to a memory block containing the pixel data for the image.
	AccelData	A pointer to a memory block containing the opaque/transparent run information. This consists of an array per scanline.
	TotalPixels	The total number of pixels in the image.
	TotalCells	The maximum number of run encodings in the image.
	XMapping	The mapping of X pixels in the compositing slice to X pixels in the

		image based on depth. This takes the form of a two dimensional array with each element containing a position and a count. The first dimension of the array corresponds to the depth of the slice and the second dimension corresponds to the X-position in the slice. This array is calculated using Grey-Codes and provides for the efficient scaling of slices with zero-order interpolation.
	YMapping	Same as *XMapping* except for Y pixels.
	CurDepth	The current slice depth which is being composited.
Methods:	IMAGE_COMPPERSPECT	Constructor.
	Clear	Clears the image to all black pixels with 0 opacity.
	GetImagePtr	Returns a pointer to the image pixel information at a specified position.
	Initialize	Initialises the mapping tables based on a scaling factor for the slices (based on depth) and a slice-inversion flag.
	SetDepth	Sets the current slice depth.
	AddPixel	Composites a new pixel into the image. The pixel is automatically resized before compositing based on the current slice depth and the position of the pixel. Also, the run encodings are automatically updated when some of the composited pixels are completely opaque.
	Run	Returns the number of pixels to skip or process from the specified position. (If the returned value is <0 then that number of pixels can be skipped, otherwise it is the amount that has to be processed.)

B.3.5 IMAGE_RGBA

Description: A 32-bit per pixel image with an alpha channel for opacity. This image is used as the output of all the rendering methods.

Members:	Data	A pointer to a memory block containing the pixel data for the image.
	TotalPixels	The total number of pixels in the image.

Methods:	IMAGE_RGBA	Constructor.
	Clear	Clears the image to all black pixels with 0 opacity.
	GetImagePtr	Returns a pointer to the image pixel information at a specified position.
	SaveBMP	Saves the image in a file using the *Microsoft Windows BMP* format.
	SaveTGA	Saves the image in a file using the *Truevision TARGA* image format.

B.3.6 MATRIX3 and MATRIX4

Description: These objects implement a C++ data-type encapsulation for a 3x3 and a 4x4 matrix. The objects actually contain arrays of the VECTOR3 and VECTOR4 objects to contain the matrix data. Numerous matrix, matrix-vector, and matrix-matrix operations are supported.

Members:	vlist	A list of vectors (VECTOR3 or VECTOR4) which comprise the rows of the matrix.

Methods:	MATRIX3	Constuctor. Default and copy constructors are provided.
	MATRIX4	
	operator *	Performs a matrix-vector (right-handed) dot product and returns a vector.
	operator +=	Adds a matrix to this one.

operator -=	Subtracts a matrix from this one.
operator *=	Multiplies this matrix with another. (right-handed)
operator *=	Multiplies this matrix with a scalar.
operator /=	Divides this matrix by a scalar.
operator []	References a particular row vector.
operator !=	Tests the inequality of two matrices.
operator ==	Tests the equality of two matrices.
Invert	Inverts the matrix. (Returns TRUE if successful.)
Transpose	Transposes the matrix.
Det	Returns the determinant of the matrix.
Adjoint	Returns the adjoint matrix for this matrix.
Clear	Clears the matrix by setting all values to one value.
Identity	Sets the matrix to the identity matrix.

B.3.7 MATRIX_MODELLING

Description: A 4x4 matrix derived from MATRIX4 which implements the modelling transformation matrix used during rendering. This matrix specifies the orientation of the volume in world-coordinates.

Members: {none}

Methods:

MATRIX_MODELLING	Constructor.
Translate	Translates the volume along each axis.
Scale	Scales the volume by a proportion along each of its axes.
RotateX	Rotates the volume around the world X-axis.
RotateY	Rotates the volume around the world Y-axis.

	RotateZ	Rotates the volume around the world Z-axis.

B.3.8 MATRIX_PROJECTION

Description: A 4x4 matrix derived from MATRIX4 which implements the projection transformation matrix used during rendering. This matrix specifies the nature of the projection of the volume in world coordinates onto the image plane (in world coordinates). The eye is always assumed to be at (0,0,-1) and the volume is centred around the origin with a unit size.

Members: {none}

Methods:	MATRIX_PROJECTION	Constructor.
	Parallel	Builds a parallel transformation matrix using the viewing frustum parameters. (In the same fashion as the *glOrtho* function in OpenGL.)
	Perspective	Builds a perspective transformation matrix using the viewing frustum parameters. (In the same fashion as the *glFrustum* function in OpenGL.)

B.3.9 MATRIX_VIEWING

Description: A 4x4 matrix derived from MATRIX4 which implements the viewing transformation matrix used during rendering. This matrix specifies the position of the eye (or camera) in the environment as a transformation from the position of the eye to (0,0,-1).

Members: {none}

Methods:	MATRIX_VIEWING	Constructor.
	Position	Specifies the x-y-z position of the eye in world coordinates.

B.3.10 PARAM_IMAGE

Description:	Contains the dimensions of the output image.	
Members:	Width	Width in pixels of the image.
	Height	Height in pixels of the image.
Methods:	{none}	

B.3.11 PARAM_OPACITIES

Description:	Contains the definitions for the opacity transfer function used during classification and rendering.	
Members:	ValueTable	A table of opacities (0-255) corresponding to different voxel values.
	GradientTable	A table of opacities (0-255) corresponding to different spatial gradient magnitudes.
	MinimumOpacity	The minimum opacity value before complete transparency is assumed.
Methods:	SetValueOpacity	Sets up the table of value-opacities my linear-interpolating between specified points in the table given their opacities.

	SetGradientOpacity	Sets up the table of gradient-opacities my linear-interpolating between specified points in the table given their opacities.

B.3.12 PARAM_LIGHTS

Description: Contains the definitions for all the lights defined in the world which illuminate the volume. All lights are assumed to be purely directional lights.

Members:	NumLights	The number of lights defined.
	Directions	An array of direction vectors for each light.
	Intensities	An array of intensity values (0 to 1) for each light. (pure white light is assumed)

Methods: {none}

B.3.13 PARAM_MATERIALS

Description: Contains the definitions of the materials corresponding to certain voxel values. (The Phong shading model is assumed.)

Members:	NumMaterials	The number of different material definitions.
	AmbientList	The ambient light emitted/reflected by each material.
	DiffuseList	The diffuse light reflected by each material.
	SpecularList	The specular light reflected by each material.
	ConeList	The shininess of each material.
	Mapping	The mapping of voxel values to material index.

Methods:	{none}	

B.3.14 PARAM_ORIENTATION

Description:	Contains the matrices specifying the orientation and viewing parameters for the volume in world coordinates.	
Members:	ModelMatrix	The modelling matrix. (MATRIX_MODELLING)
	ViewMatrix	The viewing matrix. (MATRIX_VIEWING)
	ProjectMatrix	The projection matrix. (MATRIX_PROJECTION)
Methods:	{none}	

B.3.15 VECTOR3 and VECTOR4

Description:	These objects implement a C++ data-type encapsulation for a 3-vector and a 4-vector. The objects actually contain the values for each element of the vectors. Numerous vector arithmetic operations are supported.	
Members:	x	X element. (First)
	y	Y element. (Second)
	z	Z element. (Third)
	w *(VECTOR4 only)*	W element. (Fourth)
Methods:	VECTOR3	Constructor. Default and copy constructors are provided.

VECTOR4	
operator *	Performs the dot product of two vectors.
operator +=	Adds a vector to this one.
operator -=	Subtracts a vector from this one.
operator *=	Multiplies this vector with a scalar.
operator *=	Performs the cross product of this vector with another.
operator *= *(VECTOR4 only)*	Performs left-hand multiplication with a 4x4 matrix.
operator /=	Divides this vector by a scalar.
operator []	Returns the specified element of the vector.
operator !=	Tests the inequality of two vectors.
operator ==	Tests the equality of two vectors.
Set	Sets the values in each element of the vector.
Normalize	Normalises the vector. (i.e. Makes it length 1.)
Normalize3 *(VECTOR4 only)*	Normalises the vector using homogenous coordinates.
Scale	Scales the vector to be a specified length.
Clear	Clears the vector by setting all the elements to a particular value.
Homogenize *(VECTOR4 only)*	Generates a 4-vector from a 3-vector by adding a homogenous coordinate.
DeHomogenize *(VECTOR3 only)*	Generates a 3-vector from a 4-vector by dividing out the homogenous coordinate.

B.3.16 VOLUME

Description: An abstraction of all three dimensional volumes in the system. This object provides the common memory functions and access functions for volumes. Objects of this type are

never directly created.

Members:	Stage	The stage at which the volume data is. (raw or classified)
	XLen	The X dimension of the volume in voxels.
	YLen	The Y dimension of the volume in voxels.
	ZLen	The Z dimension of the volume in voxels.

Methods:	VOLUME	Constructor.
	Access	A virtual function which returns a memory pointer to the allocated volume data.
	operator =	A copy constructor for the data in the VOLUME object only.
	SetSize	Sets the dimensions of the volume.
	GetXSize	Returns the X dimension of the volume.
	GetYSize	Returns the Y dimension of the volume.
	GetZSize	Returns the Z dimension of the volume.

B.3.17 VOLUME_RAW

Description: An encapsulation of a raw three dimensional array of volume data. The data may be in one of two states: (1) Original raw form; and (2) an optimised and filtered form. Various operations are available for parsing volume data files and for optimising these datasets once they are parsed.

Members:	RawData	A pointer to a memory block containing the volume data.
	RawHeader	A structure describing the nature of the volume: • dimensions, • physical size, and • size of voxel elements.

Methods:	VOLUME_RAW	Constructor.
	Parse	Parses a file of raw volume information using the information provided to the *Configure* method (below).
	Load	Loads a previously saved VOLUME_RAW object.
	Save	Saves a VOLUME_RAW object to a file.
	Configure	Configures the volume data parser given: • dimensions of the volume, and • bytes per voxel.
	Optimize	Optimises the range of values in the volume and converts each voxel to a byte. Accepts: • Minimum and maximum values in original data. • Little-endian, Big-endian flag. • Signed integer flag. • Bit mask (used before optimising).
	BuildHistogram	Builds a histogram of the raw data by generating an array (each element represents one possible voxel value) where each element contains the number of voxels with that particular value.
	GetBytesPerVoxel	Returns the bytes per voxel in the volume.

B.3.18 VOLUME_RLE

Description:	An encapsulation of an RLE compressed data volume. The volume may be in either a unclassified or a classified state. Various methods are provided to compress, classify, and render the data. This object contains the classic implementation of the Shear-Warp Factorisation algorithm.	
Members:	RleHeader	A structure describing the nature of the volume data:

- dimensions of the volume,
- bytes per voxel,
- raw size of the volume,
- raw size of the stick list (described below),
- number of opaque voxels,
- low and high cutoff thresholds.

RawData	A pointer to a memory block containing the volume data in X-Y-Z order.
Sticks	A two dimensional array containing pointers into the volume data where each new X-stick begins. This array references the beginning of each one dimensional list of voxels beginning on the X-minimum side of the volume. (This allows for efficient leaping to certain positions as each scanline is not necessarily compressed by the same amount.)
RawY	A transposed copy of the volume in *RawData* using ZXY order.
SticksY	The stick list for *RawY*.
RawZ	A transposed copy of the volume in *RawData* using YZX order.
SticksZ	The stick list for *RawZ*.
ShadeTable	A multi-dimensional array with dimensions for material number, light number, and normal value, and each element contains an RGB triple. This table allows for very efficient rendering by performing all the expensive Phong shading computations once before-hand.
{Parameters}	A copy of all the parameter objects currently being used.
FinalMatrix	The result of merging the modelling, viewing, and projection matrices with the permutation matrix.
TraversalDir	The direction of slice traversal through the volume.
ShearI	The shearing factor in the transformed X direction.
ShearJ	The shearing factor in the transformed Y direction.
TransI	The translation factor in the transformed X direction.
TransJ	The translation factor in the transformed Y direction.
WarpMatrix	The image warping matrix which converts the intermediate compositing image to the final image.

	CompImage	The intermediate compositing image. (IMAGE_COMPPARALLEL)
	FinalImage	The final image. (IMAGE_RGBA)

Methods:	VOLUME_RLE	Constructor.
	Load	Loads a saved VOLUME_RLE object from a file.
	Save	Saves a VOLUME_RLE object to a file.
	Construct	Constructs the data in the VOLUME_RLE object from a specified VOLUME_RAW object. The low and high threshold values must be supplied as well. This will then perform the initial RLE compression on the volume.
	Classify	Classifies the compressed RLE voxel data given a PARAM_OPACITIES object.
	Render	Renders a parallel projection of the RLE data using the Shear-Warp algorithm. All parameter objects must be passed for the rendering to occur. If certain parameters change between renderings then the affected internal tables will be re-computed.
	GetHeader	Returns the *RleHeader* data structure.
	GetImage	Returns a reference to the *FinalImage* object where the rendered object will be.
	MemoryUsed	Reports the amount of runtime memory currently being used by this object.

B.3.19 VOLUME_OCTREE

Description: An encapsulation of an octree compressed data volume. The volume may be in either a unclassified or a classified state. Various methods are provided to compress, classify, and render the data. This object contains most of the algorithms presented in this dissertation.

Members:	OctHeader	A structure describing the nature of the volume data: • dimensions of the volume, • bytes per voxel, • raw size of the volume, • number of nodes in the octree, • number of opaque voxels, • low and high cutoff thresholds, • minimum and maximum values and gradients in the octree, • the type of leaf-node compression used, and • average value, gradient, and normal in the entire volume.
	RawData	A pointer to a memory block containing the raw volume data which is referenced by the octree data structure.
	BaseOctree	The basic octree data structure which is constructed and used for transmission and classification.
	RenderOctree	The secondary octree data structure (structurally the same as *BaseOctree*) constructed during classification and used for rendering.
	TraversalList	Parallel rendering node traversal order. This is set up prior to rendering as it depends on traversal direction and shearing factors.
	PrspTraversalList	Possible perspective rendering node traversal orders. This list is calculated before rendering based on viewing direction, however further reduction is performed during rendering based on node shearing and scaling.
	LimitData	A data limit representing the amount of valid data in the *BaseOctree* structure and then the *RawData* memory. This value will increase as new data is transmitted until it is equal to the total size of the octree and all the raw data. All classification and rendering algorithms check this value to know when to approximate missing data.
	ShadeTable	A multi-dimensional array with dimensions for material number, light number, and normal value, and each element contains an RGB triple. This table allows for very efficient rendering by

		performing all the expensive Phong shading computations once before-hand.
	{Parameters}	A copy of all the parameter objects currently being used.
	FinalMatrix	The result of merging the modelling, viewing, and projection matrices with the permutation matrix.
	TraversalDir	The direction of slice traversal through the volume.
	ShearI	The shearing factor in the transformed X direction.
	ShearJ	The shearing factor in the transformed Y direction.
	TransI	The translation factor in the transformed X direction.
	TransJ	The translation factor in the transformed Y direction.
	WarpMatrix	The image warping matrix which converts the intermediate compositing image to the final image.
	CompImage	The intermediate compositing image. (IMAGE_COMPPARALLEL or IMAGE_COMPPERSPECT)
	FinalImage	The final image. (IMAGE_RGBA)
	OctCache	The octree node cache array used during classification.
Methods:	VOLUME_OCTREE	Constructor.
	Load	Loads a saved VOLUME_OCTREE object from a file.
	Save	Saves a VOLUME_OCTREE object to a file.
	Construct	Constructs the data in the VOLUME_OCTREE object from a specified VOLUME_RAW object. The low and high threshold values must be supplied as well as the maximum octree depth. This will then perform the initial octree compression on the volume.
	Classify	Classifies the compressed octree voxel data given a PARAM_OPACITIES object.
	Render	Renders a parallel or perspective projection of the octree data using the Shear-Warp algorithm. All parameter objects must be passed for the rendering to occur. If certain parameters change between renderings then the affected internal tables will be re-computed.
	GetHeader	Returns the *OctHeader* data structure.

GetImage	Returns a reference to the *FinalImage* object where the rendered object will be.
MemoryUsed	Reports the amount of runtime memory currently being used by this object.

Appendix C

Graphs

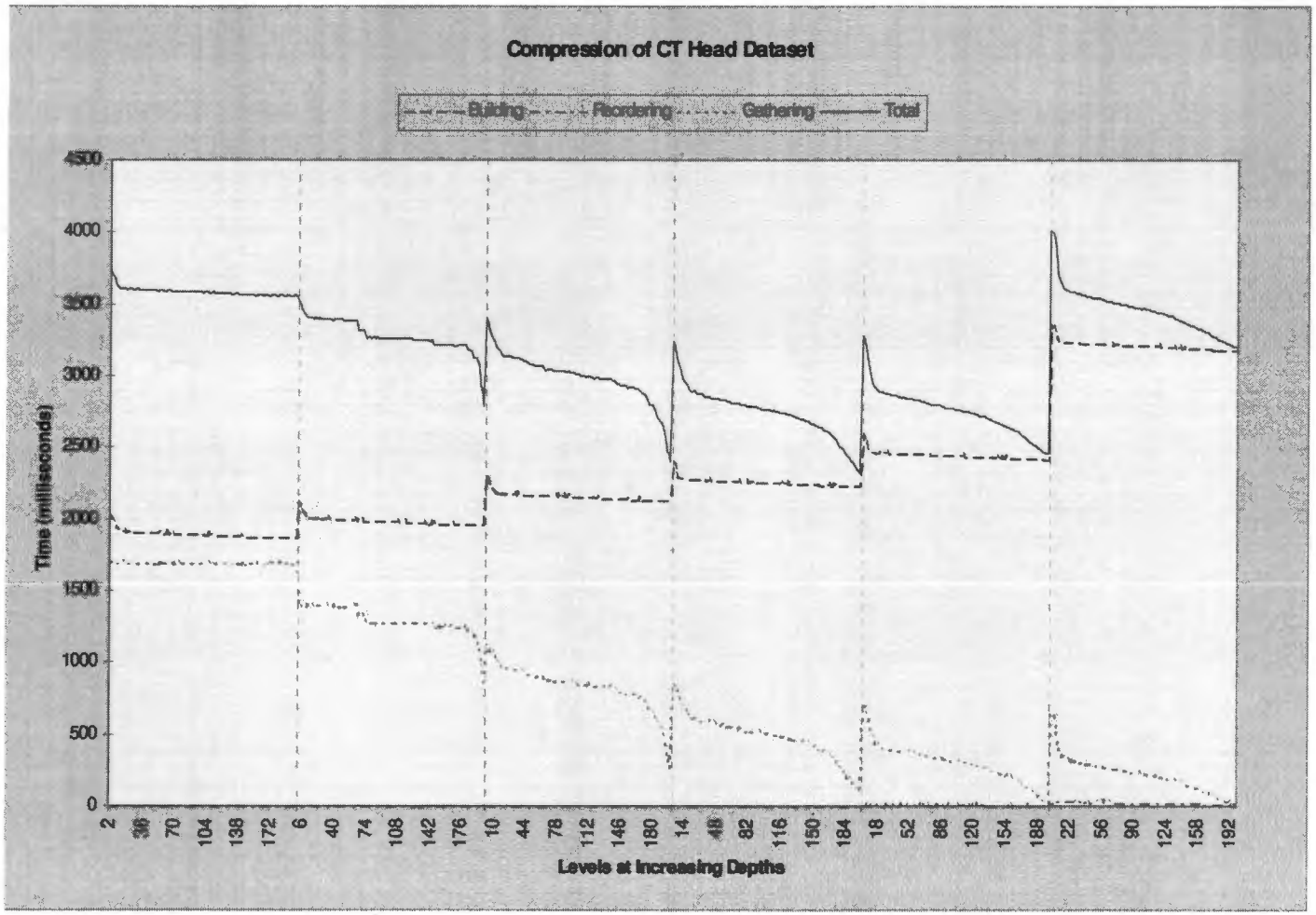


Figure C.1 - Performance of the octree compression algorithm (without node compression) on the CT Head dataset at six different depths and for numerous levels.

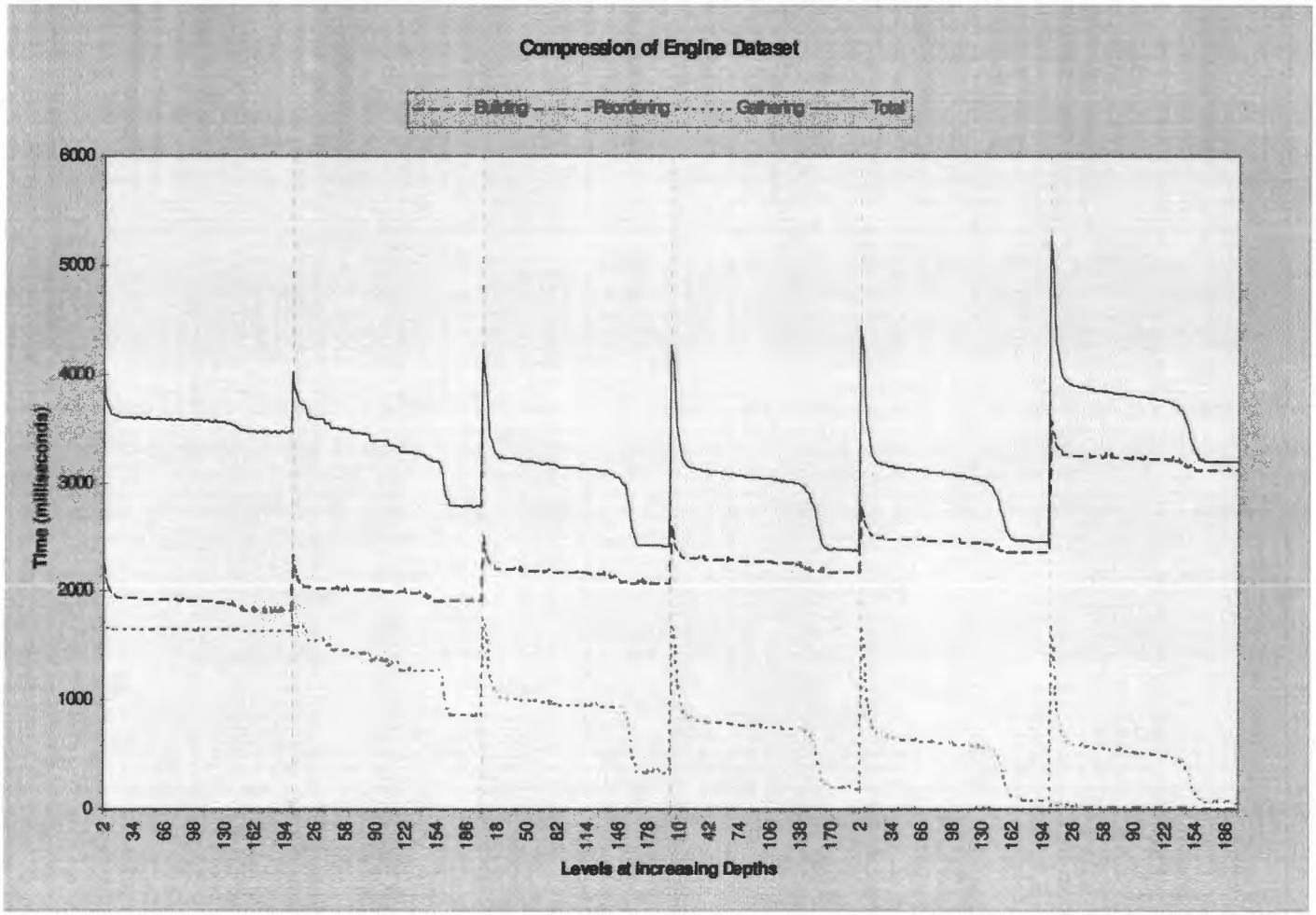


Figure C.2 - Performance of the octree compression algorithm (without node compression) on the Engine dataset at six different depths and for numerous levels.

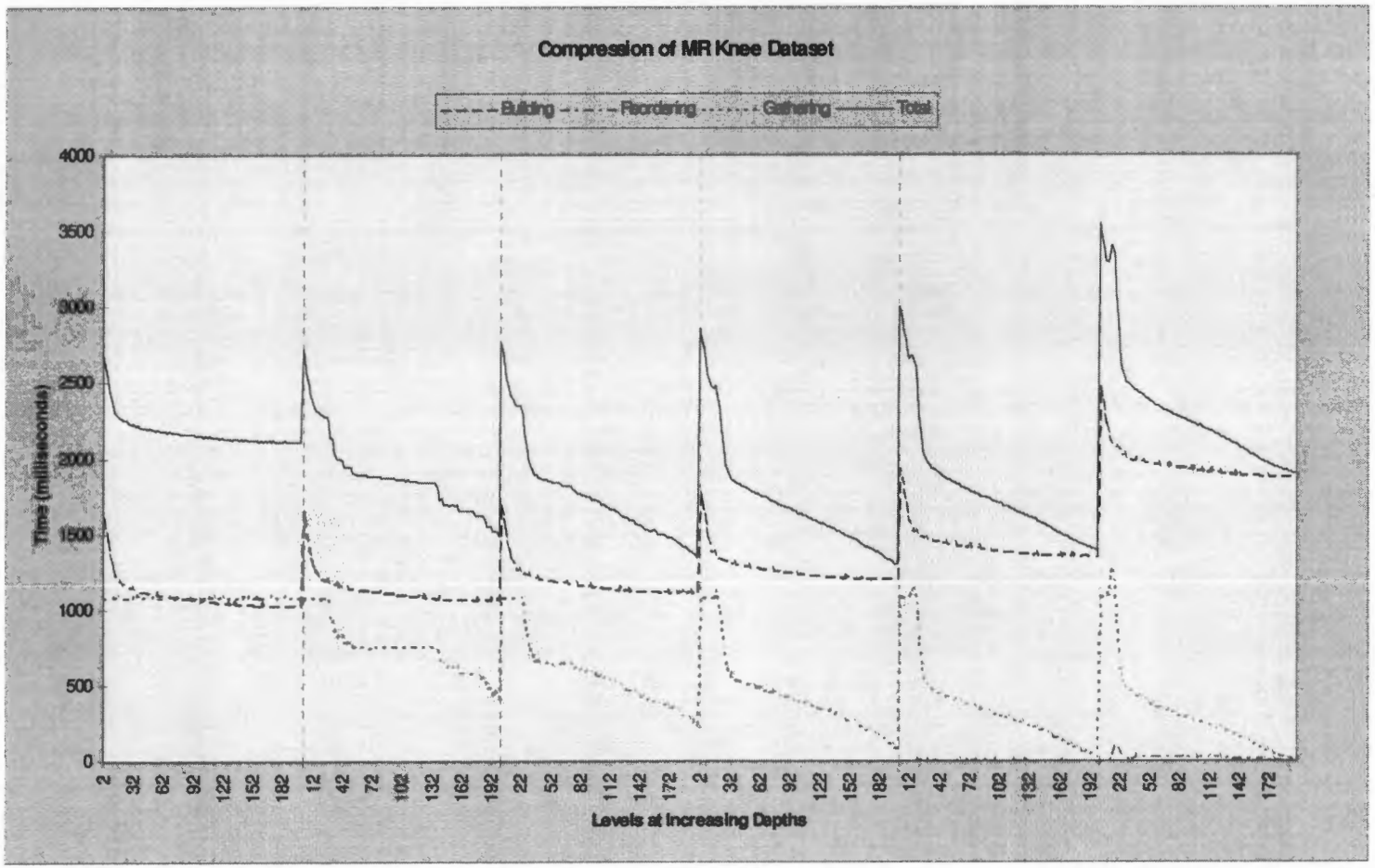


Figure C.3 - Performance of the octree compression algorithm (without node compression) on the MR Knee dataset at six different depths and for numerous levels.

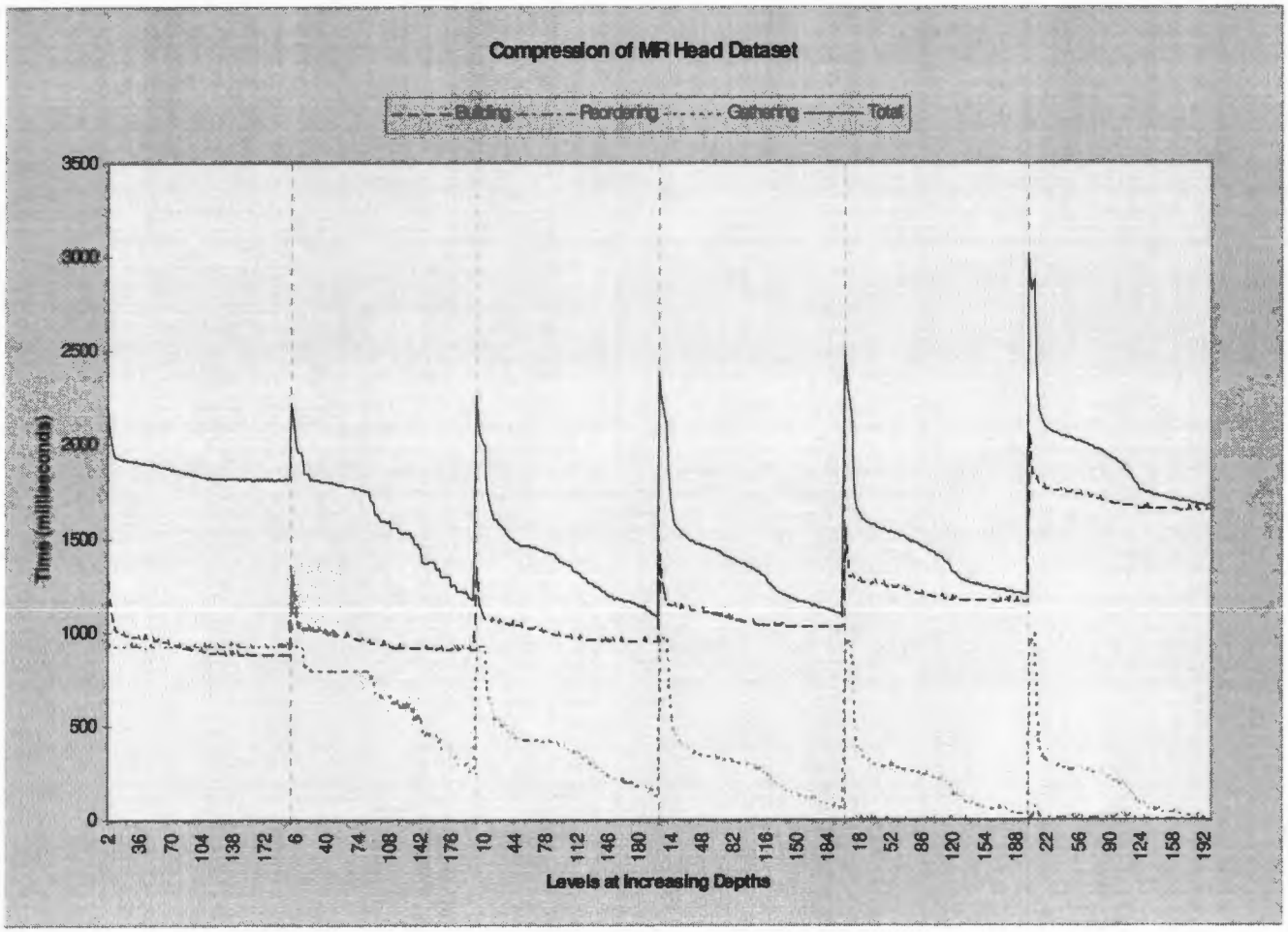


Figure C.4 - Performance of the octree compression algorithm (without node compression) on the MR Head dataset at six different depths and for numerous levels.

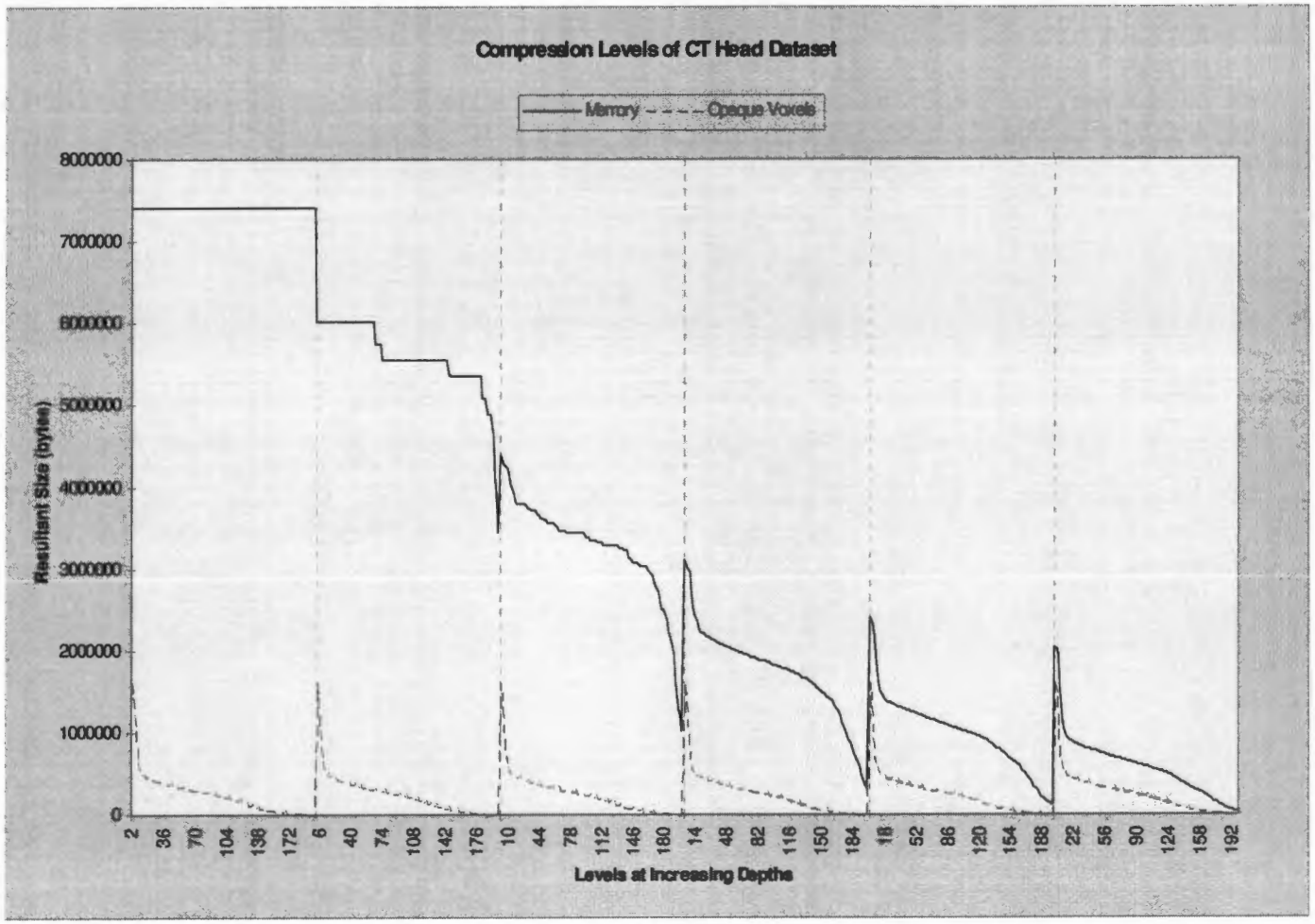


Figure C.5 - Compression results for the octree compression algorithm (without node compression) for the CT Head dataset.

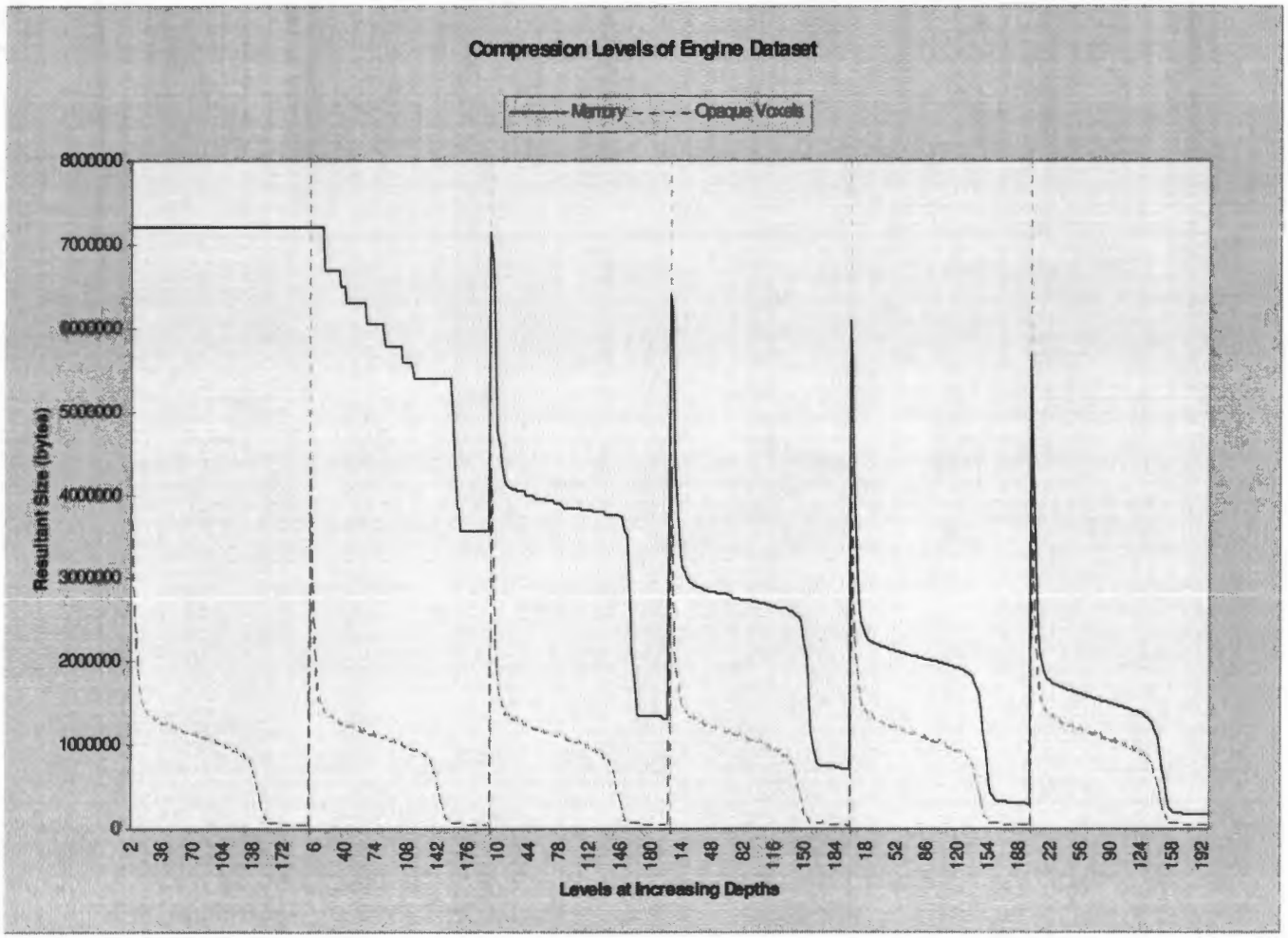


Figure C.6 - Compression results for the octree compression algorithm (without node compression) for the Engine dataset.

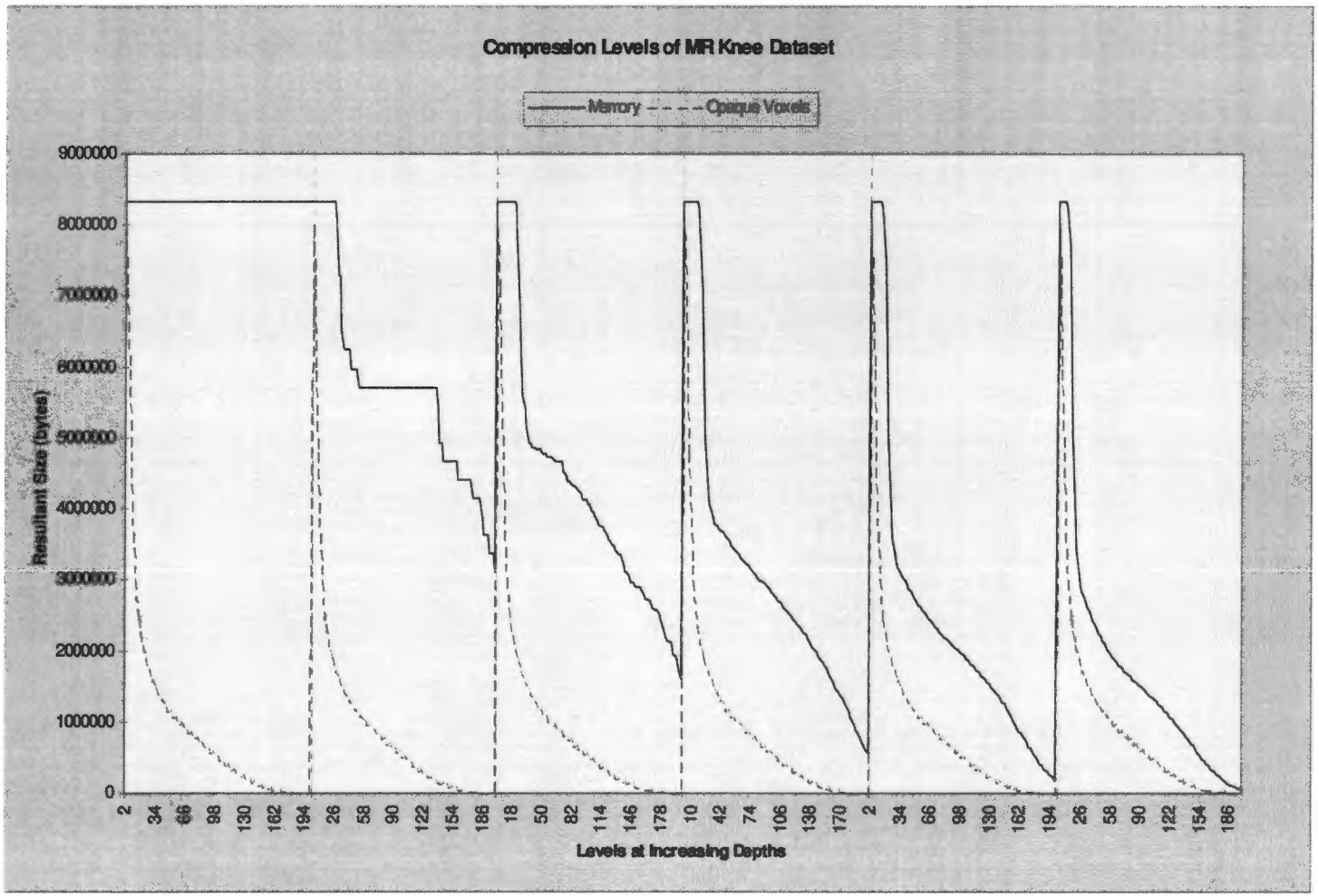


Figure C.7 - Compression results for the octree compression algorithm (without node compression) for the MR Knee dataset.

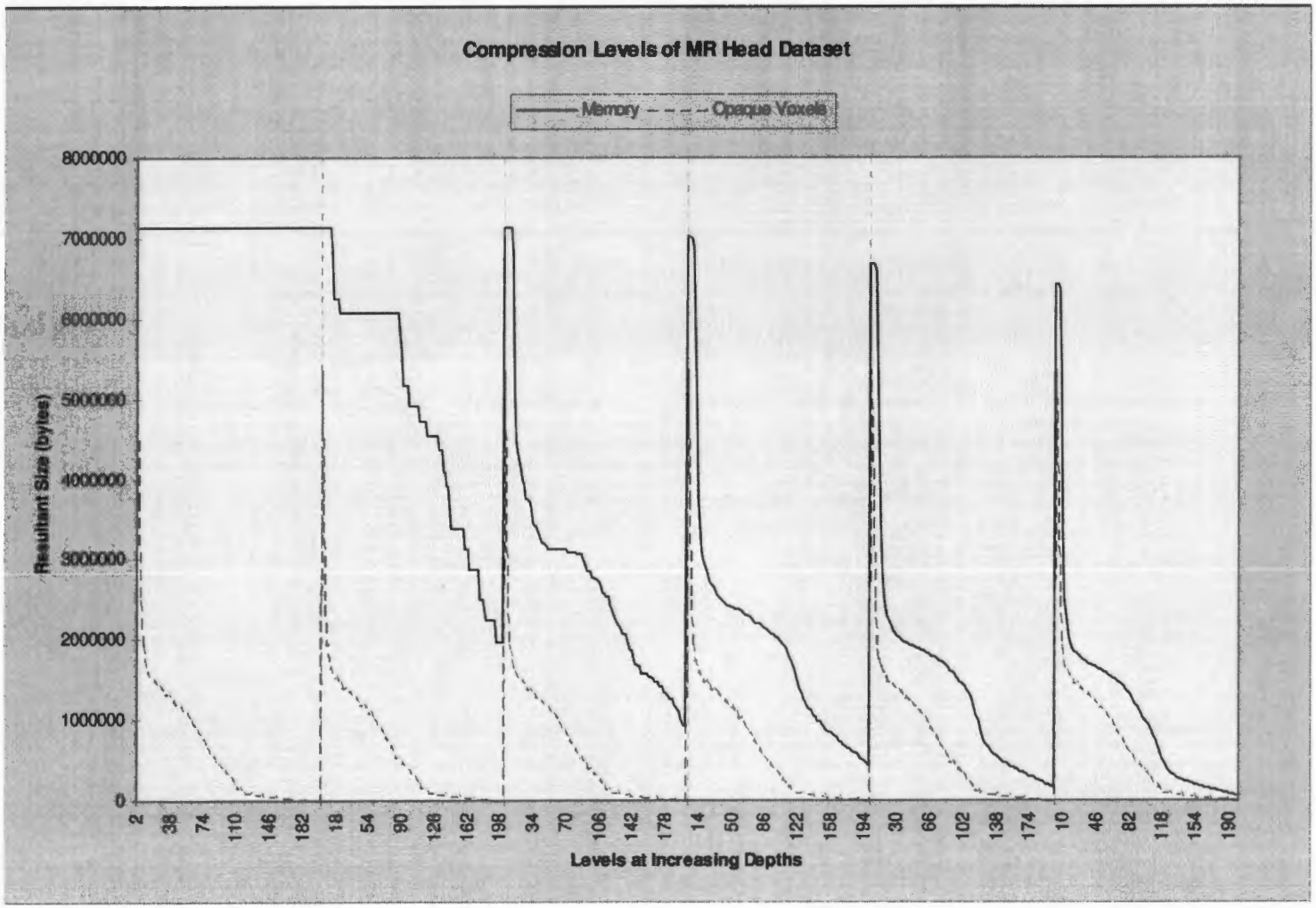


Figure C.8 - Compression results for the octree compression algorithm (without node compression) for the MR Head dataset.

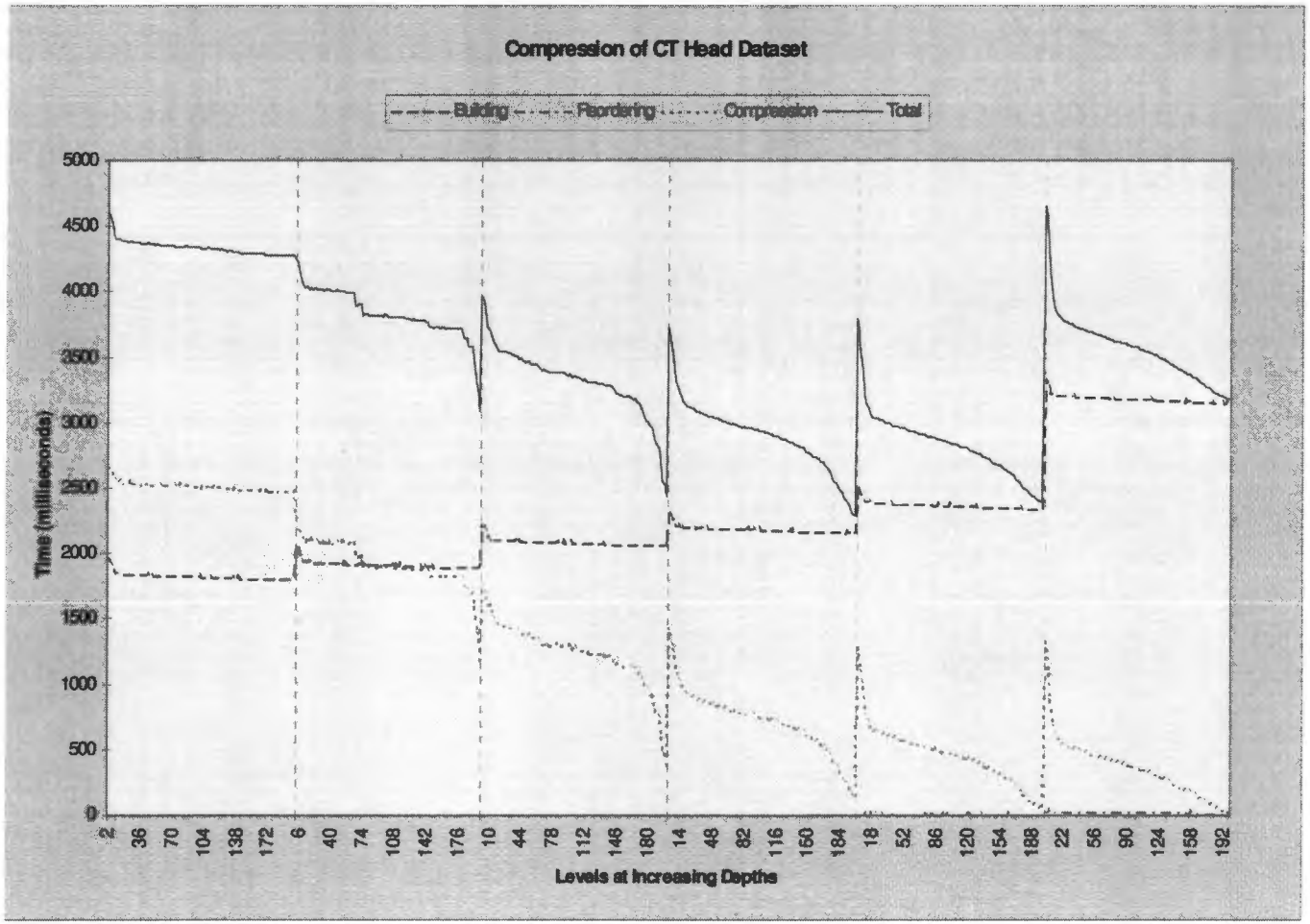


Figure C.9 - Performance of the octree compression algorithm (with node compression) on the CT Head dataset at six different depths and for numerous levels.

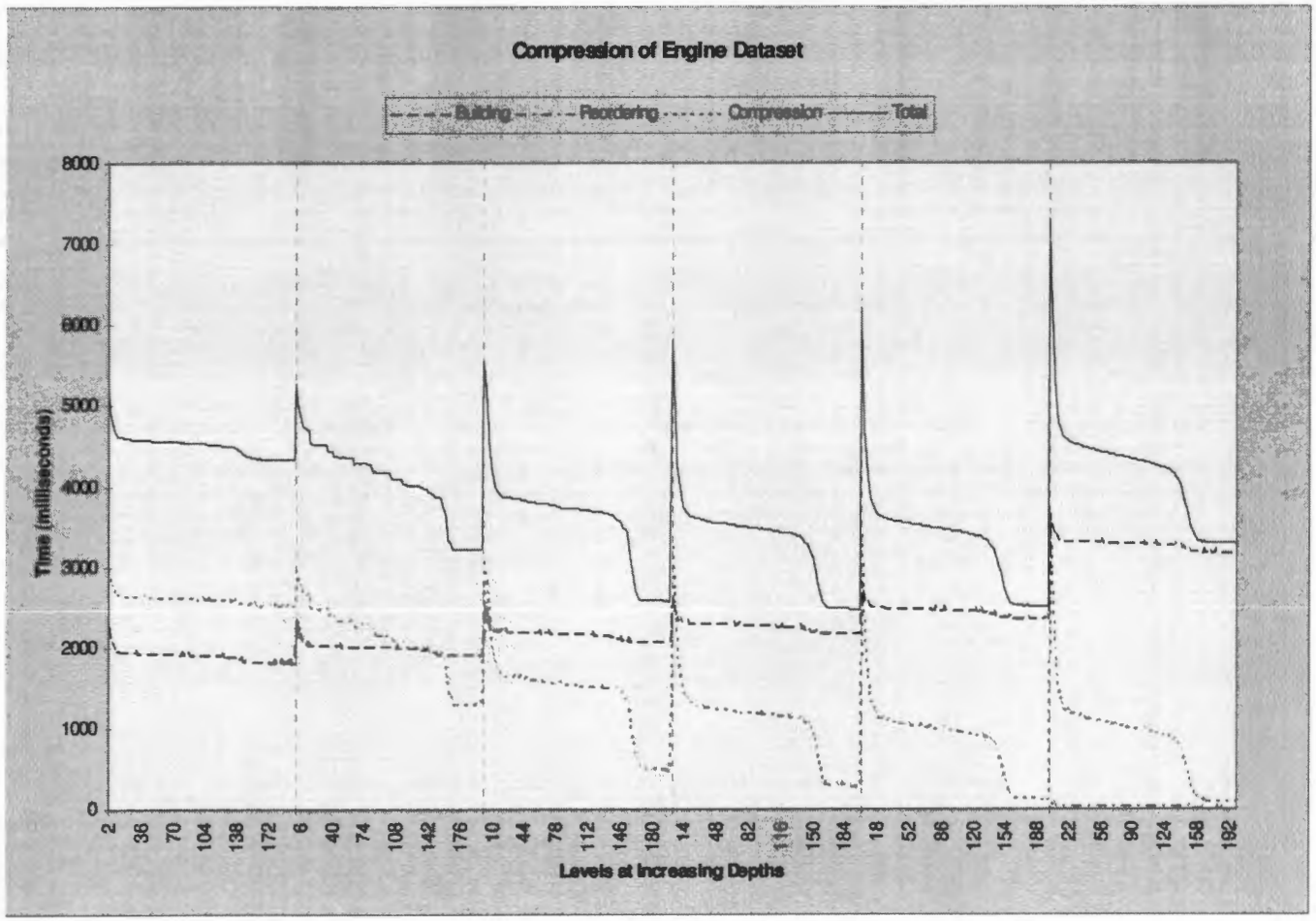


Figure C.10 - Performance of the octree compression algorithm (with node compression) on the Engine dataset at six different depths and for numerous levels.

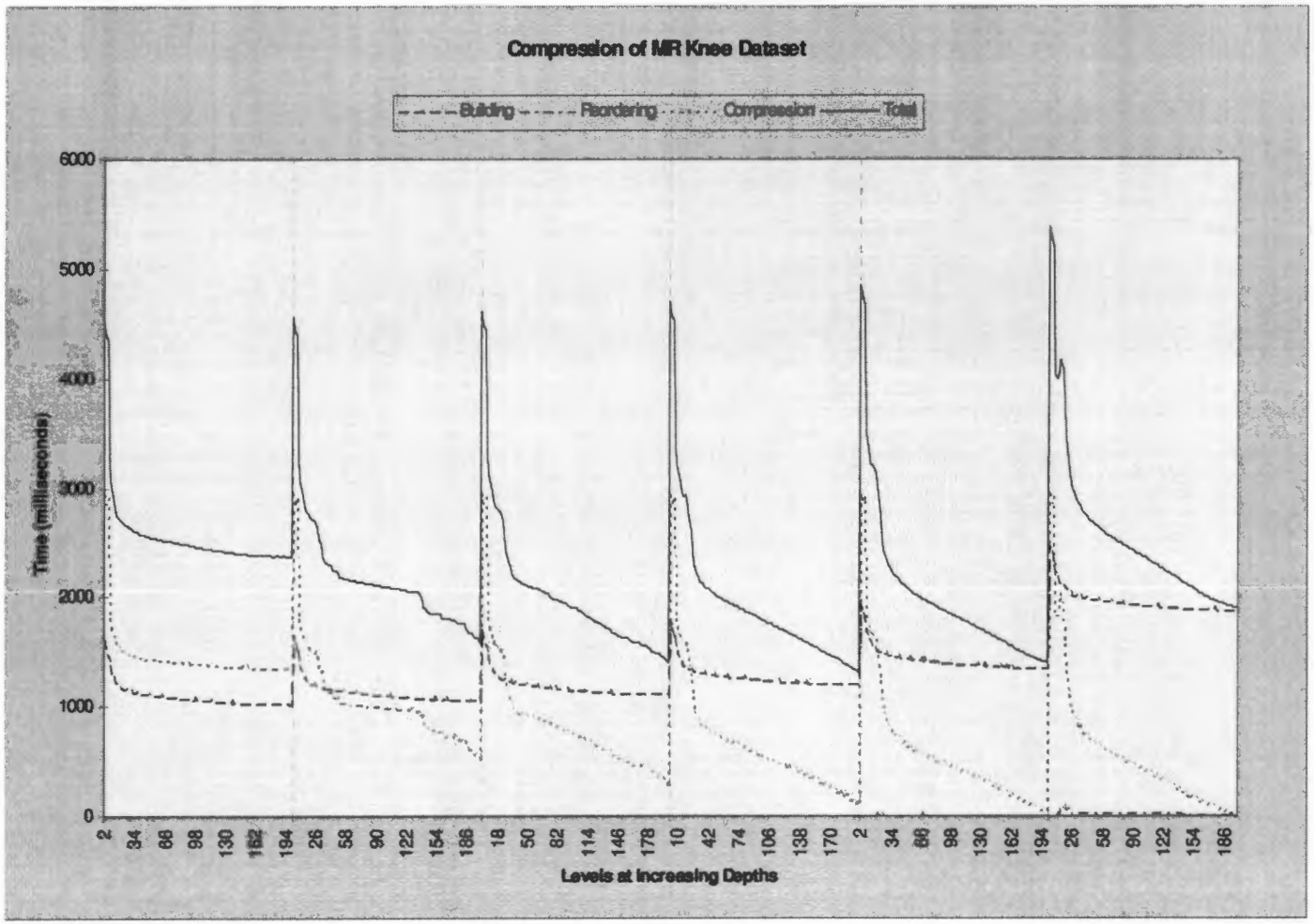


Figure C.11 - Performance of the octree compression algorithm (with node compression) on the MR Knee dataset at six different depths and for numerous levels.

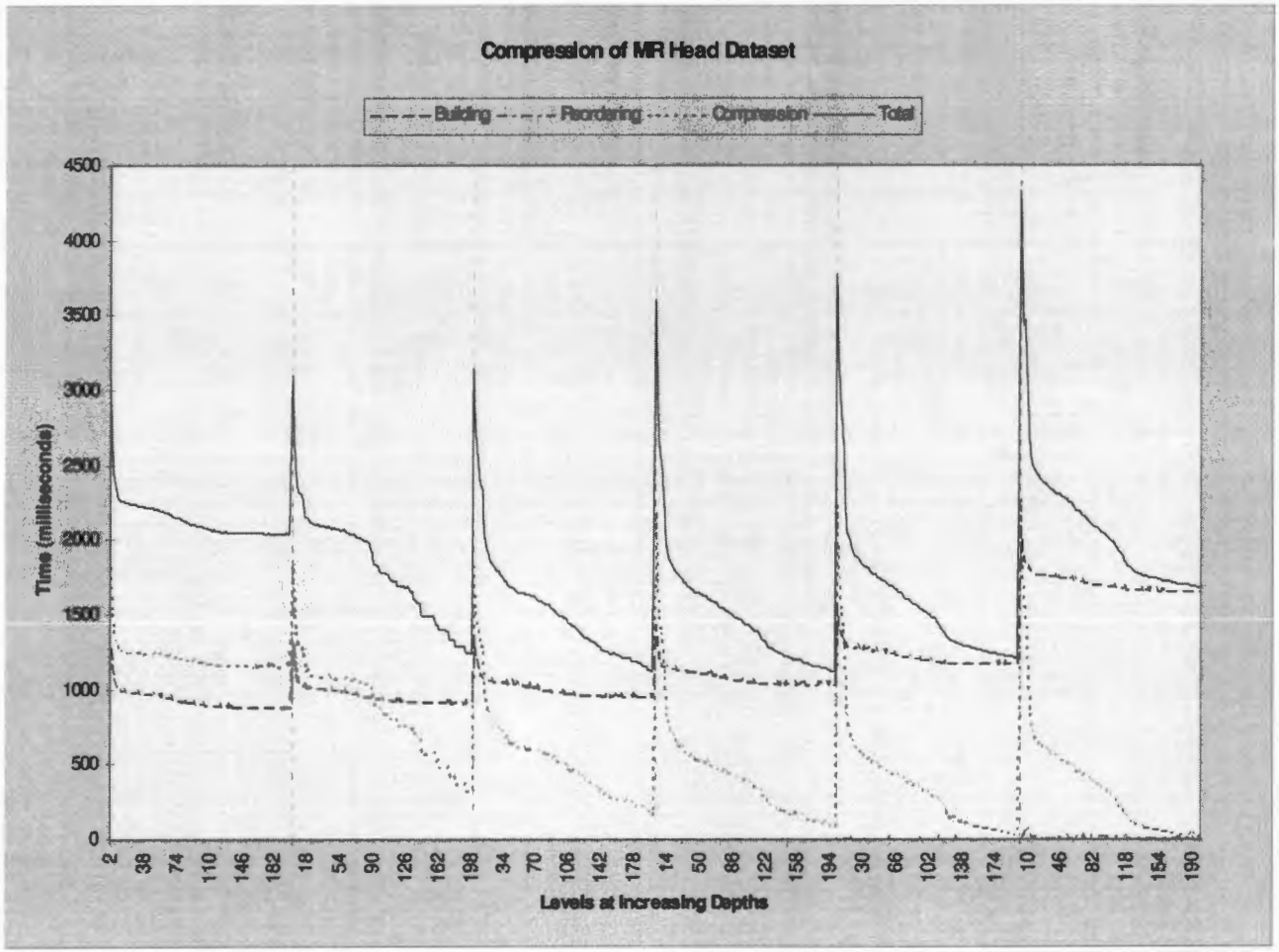


Figure C.12 - Performance of the octree compression algorithm (with node compression) on the MR Head dataset at six different depths and for numerous levels.

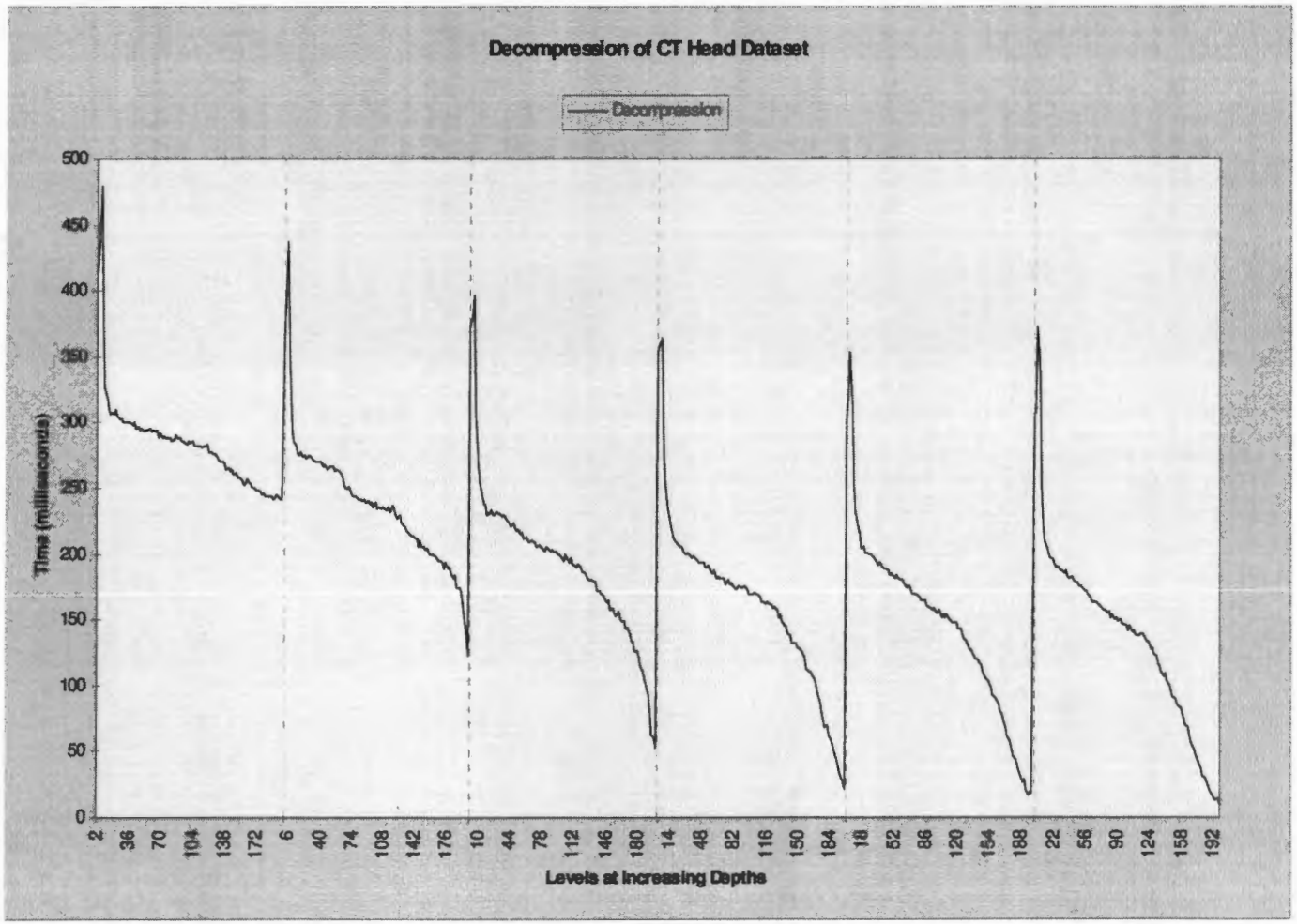


Figure C.13 - Decompression performance of the octree algorithm (with node compression) on the CT Head dataset.

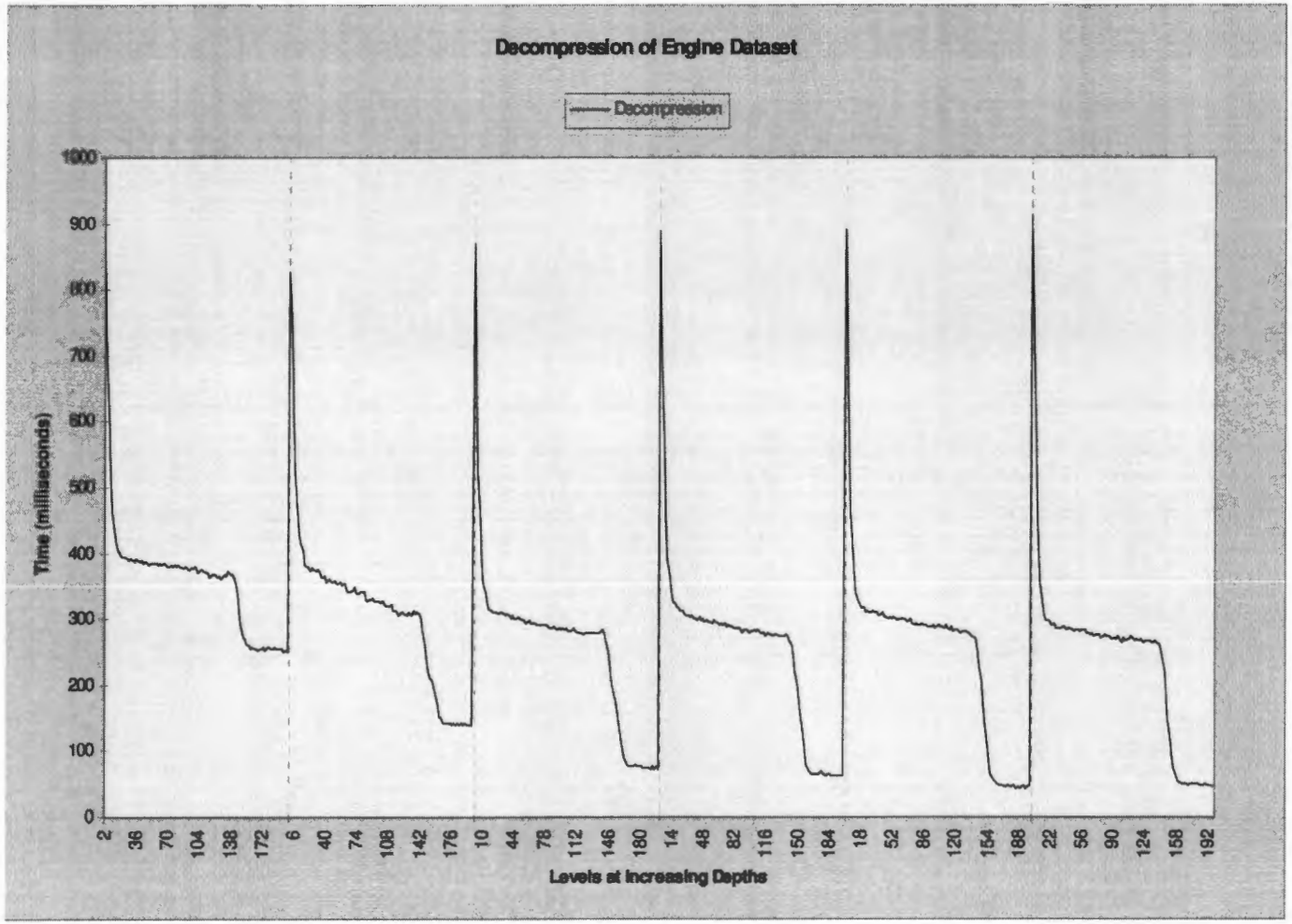


Figure C.14 - Decompression performance of the octree algorithm (with node compression) on the Engine dataset.

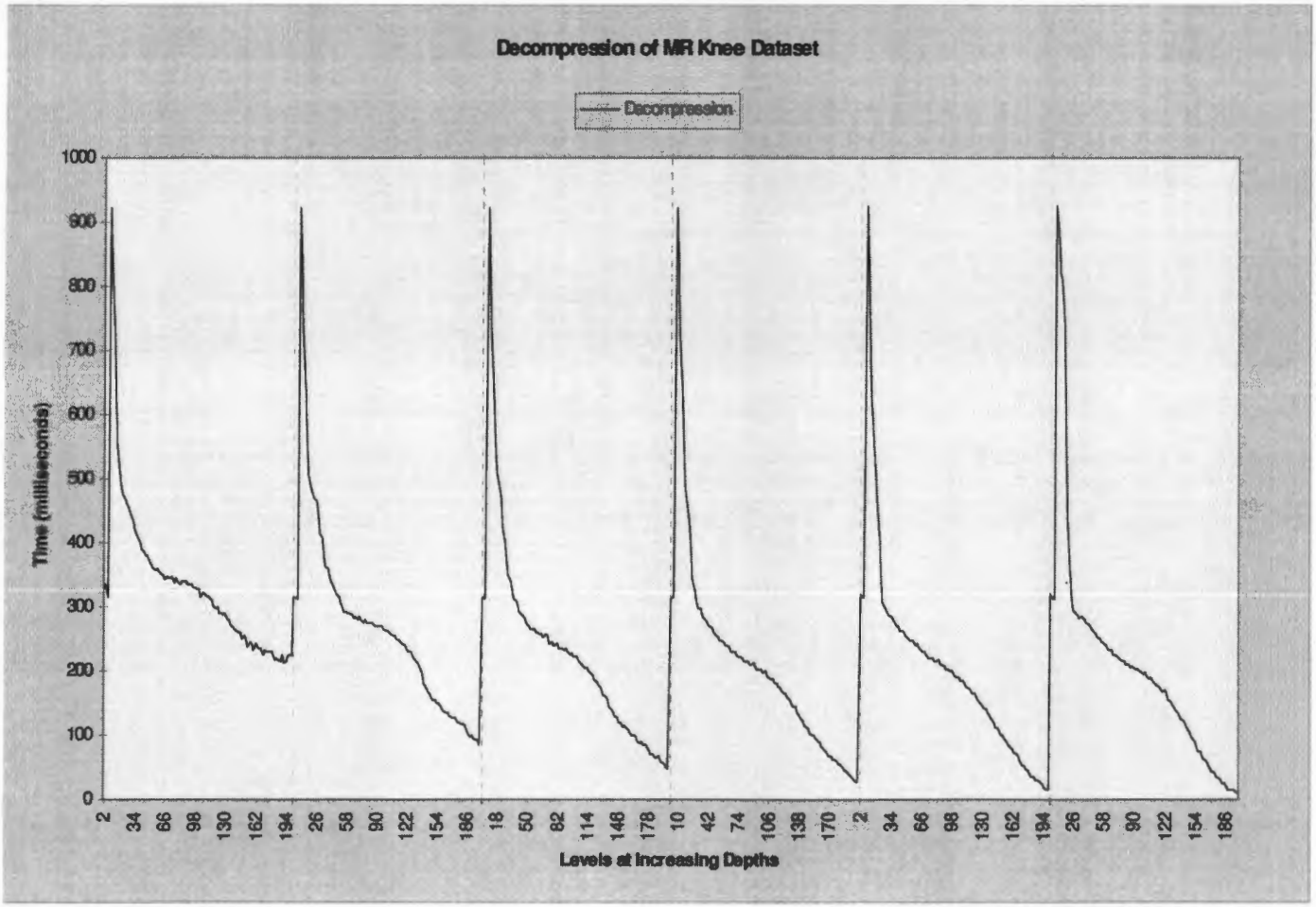


Figure C.15 - Decompression performance of the octree algorithm (with node compression) on the MR Knee dataset.

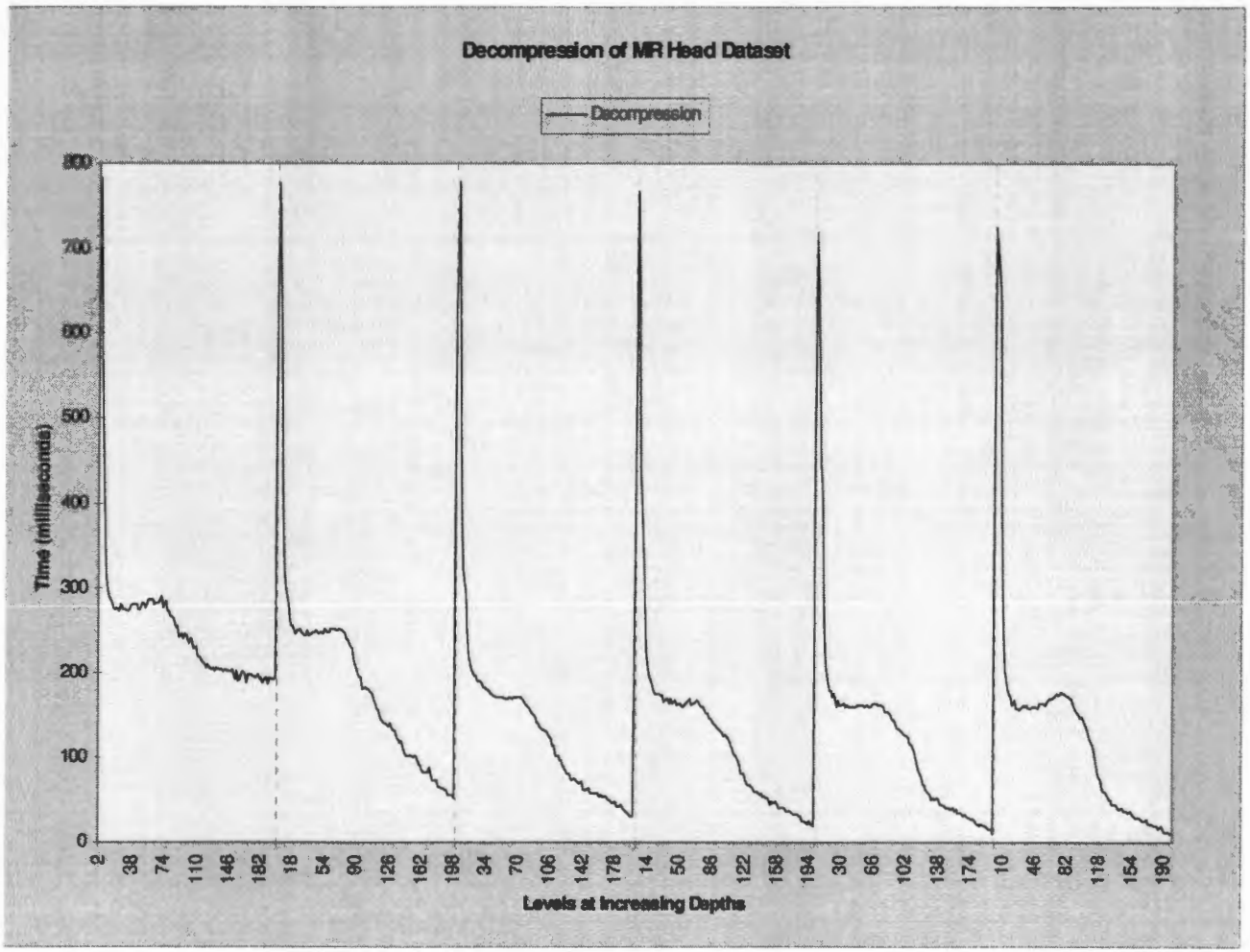


Figure C.16 - Decompression performance of the octree algorithm (with node compression) on the MR Head dataset.

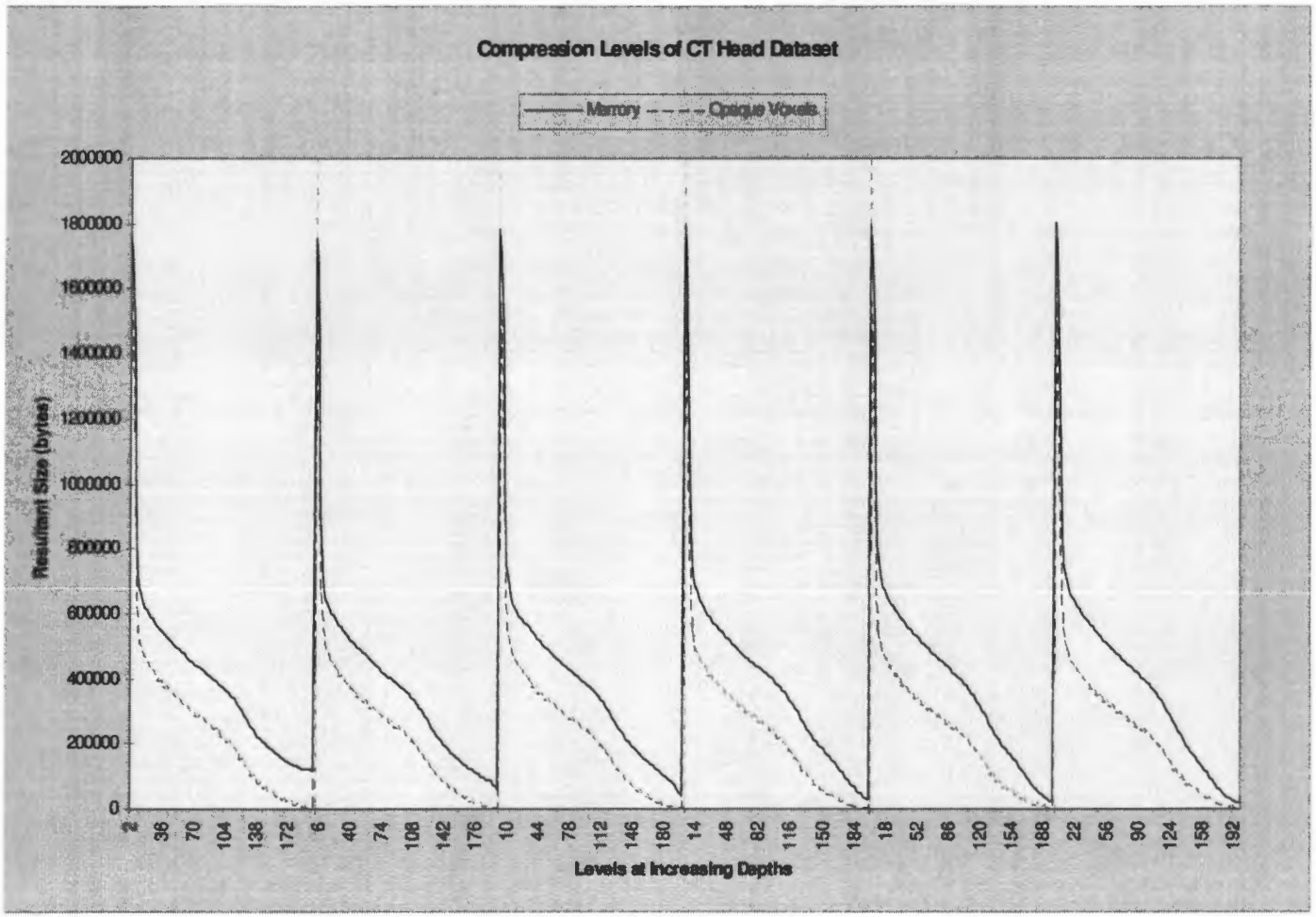


Figure C.17 - Compression results for the octree compression algorithm (with node compression) for the CT Head dataset.

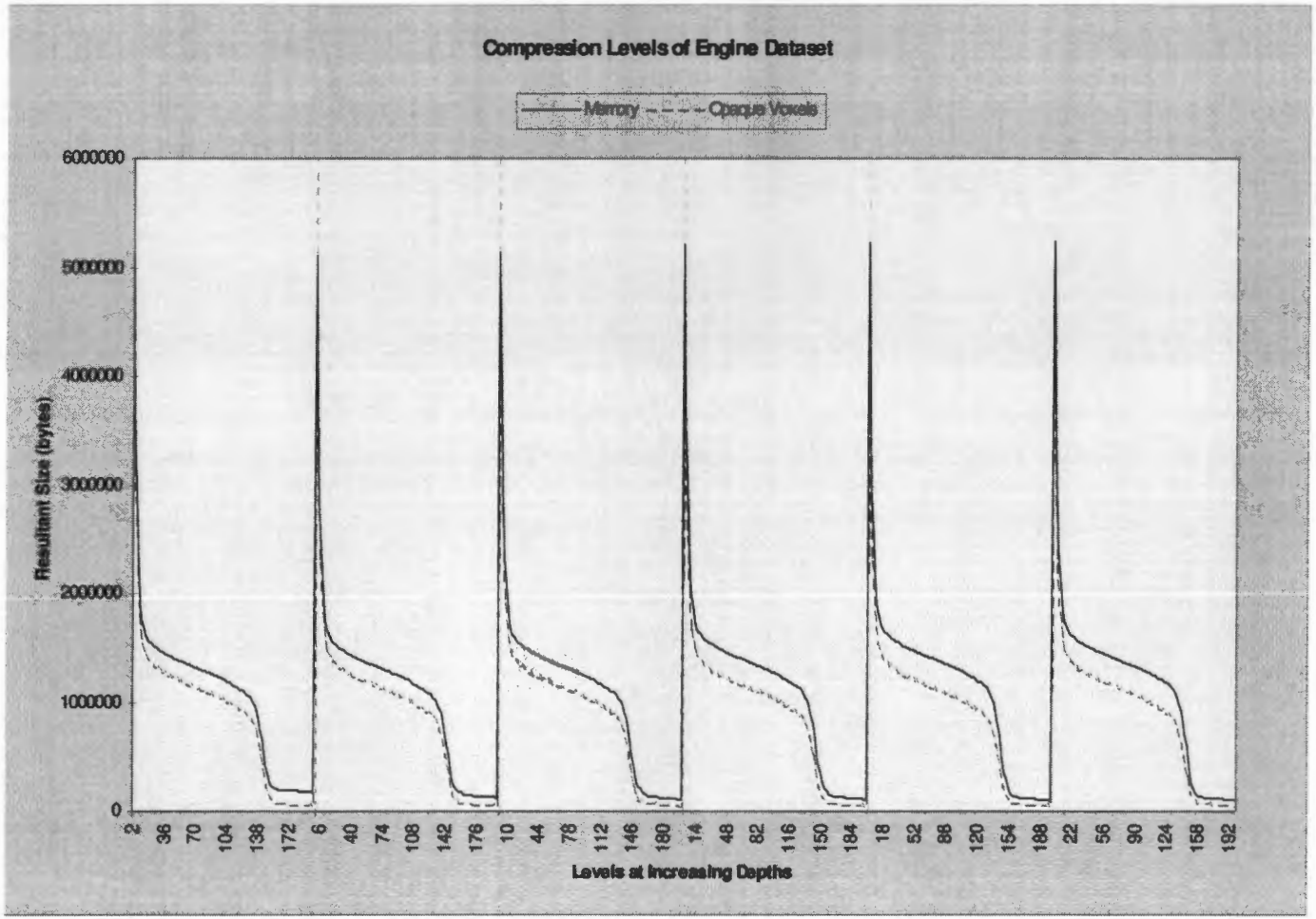


Figure C.18 - Compression results for the octree compression algorithm (with node compression) for the Engine dataset.

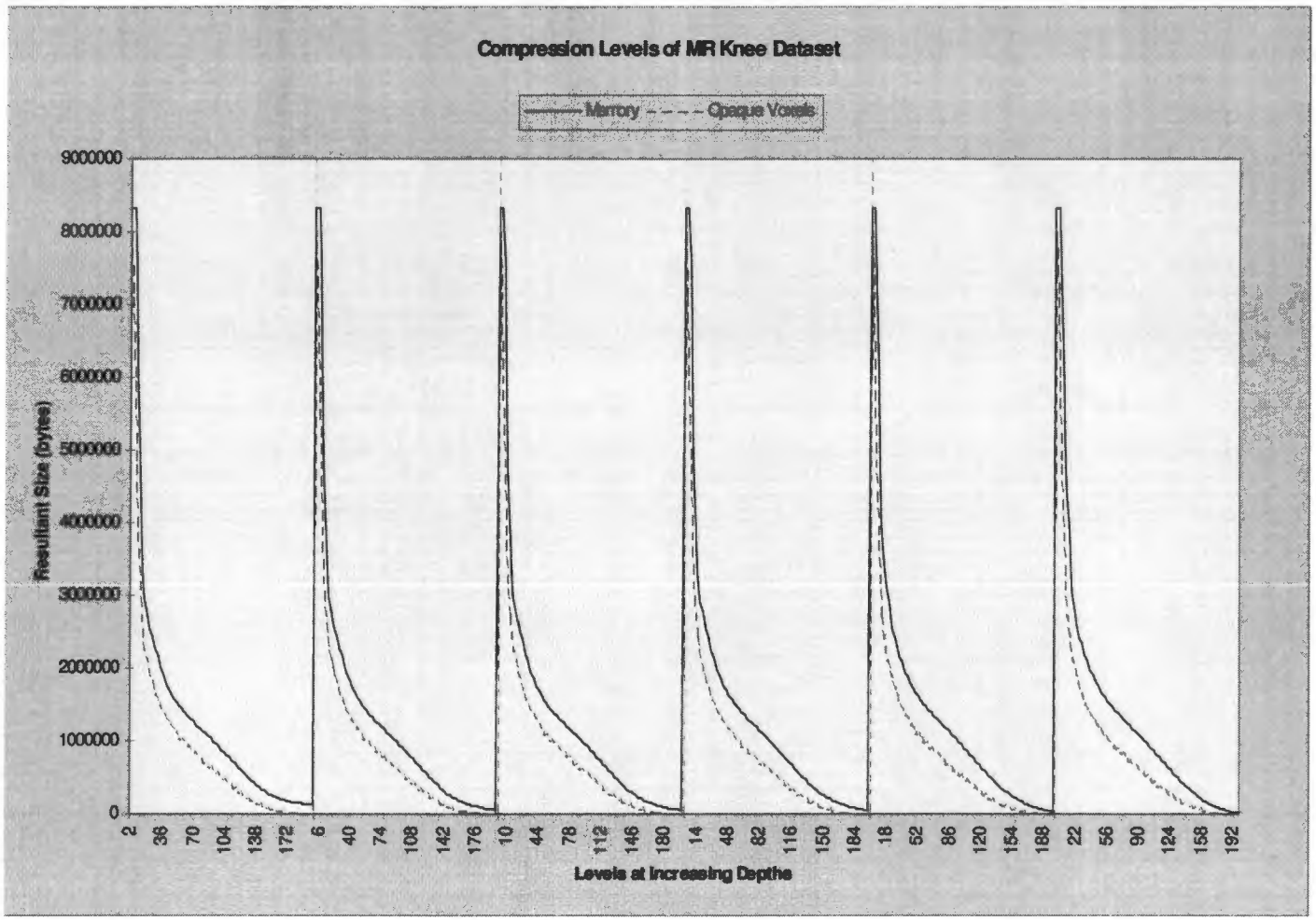


Figure C.19 - Compression results for the octree compression algorithm (with node compression) for the MR Knee dataset.

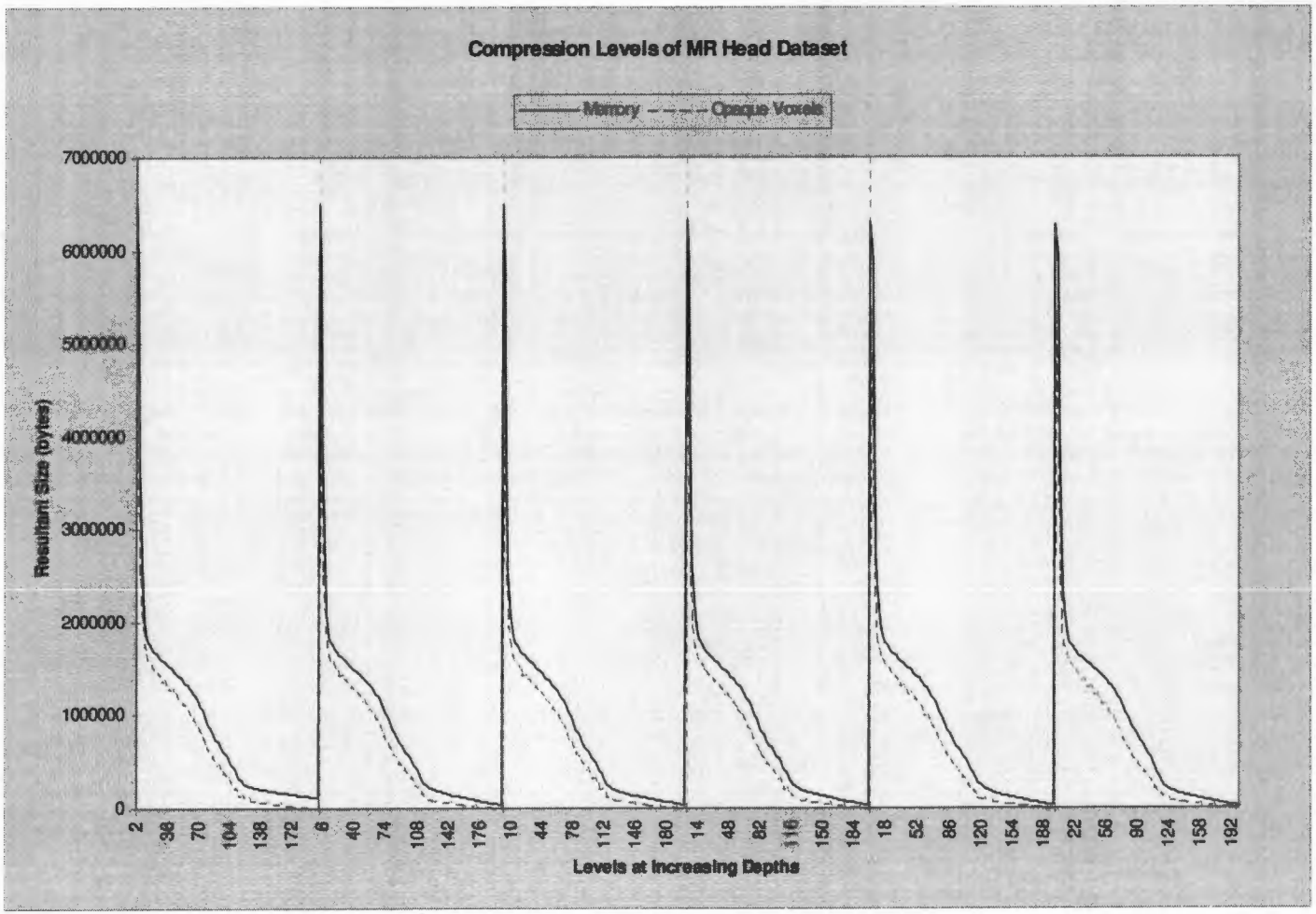


Figure C.20 - Compression results for the octree compression algorithm (with node compression) for the MR Head dataset.

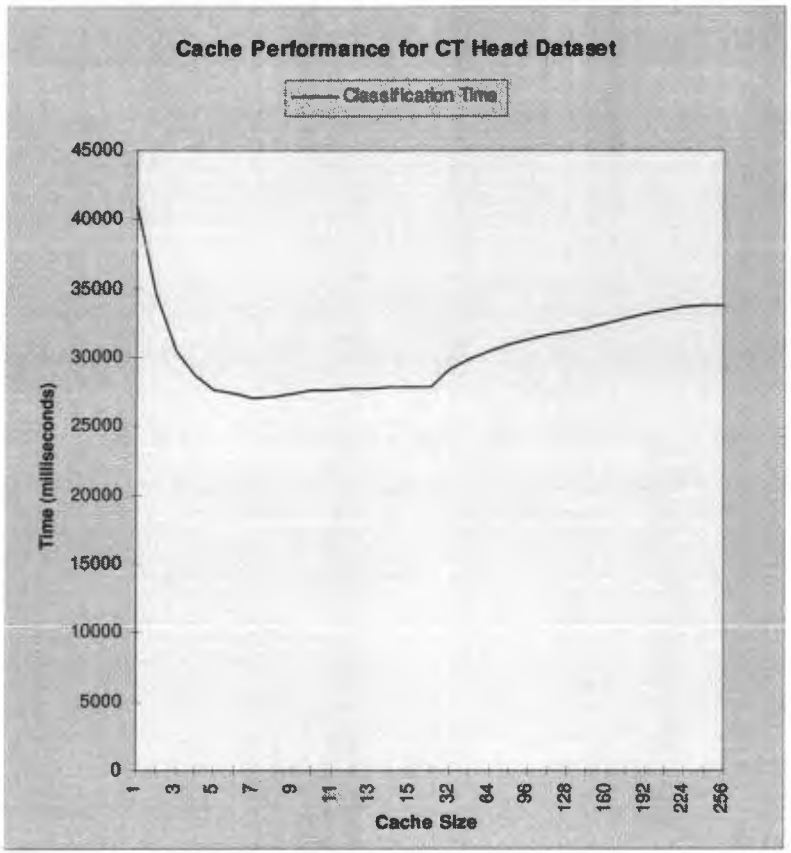


Figure C.21 - Classification performance on the CT Head dataset for varying cache sizes.

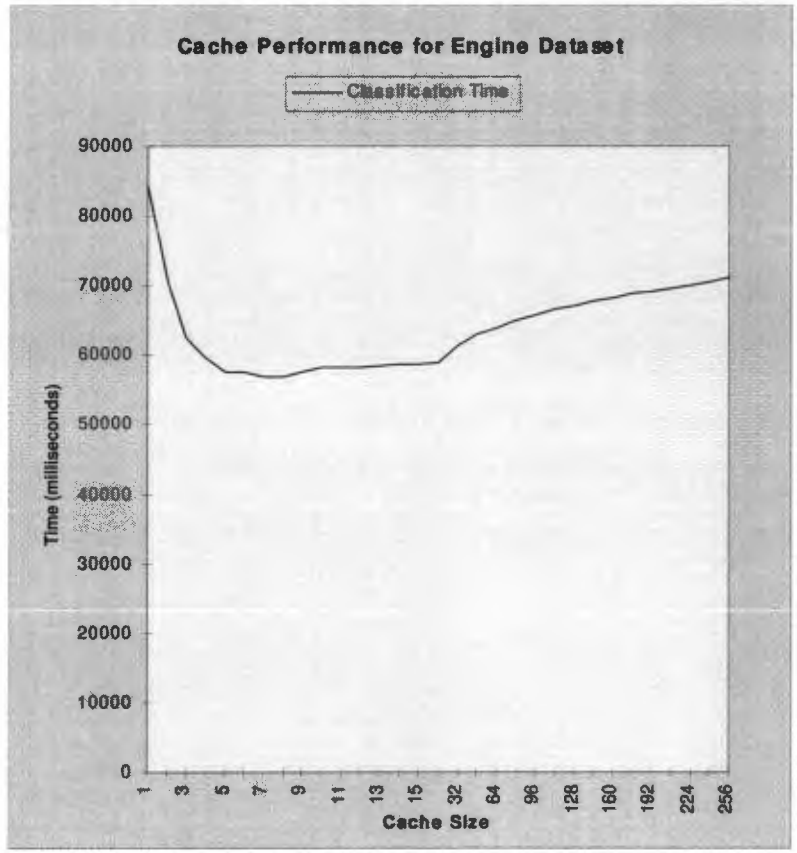


Figure C.22 - Classification performance on the Engine dataset for varying cache sizes.

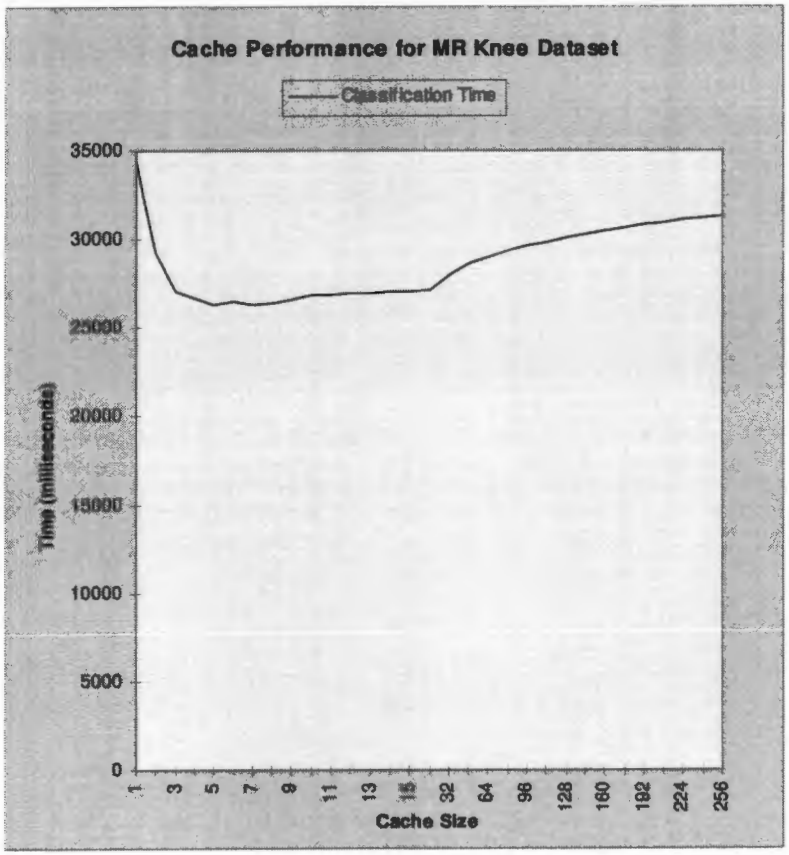


Figure C.23 - Classification performance on the MR Knee dataset for varying cache sizes.

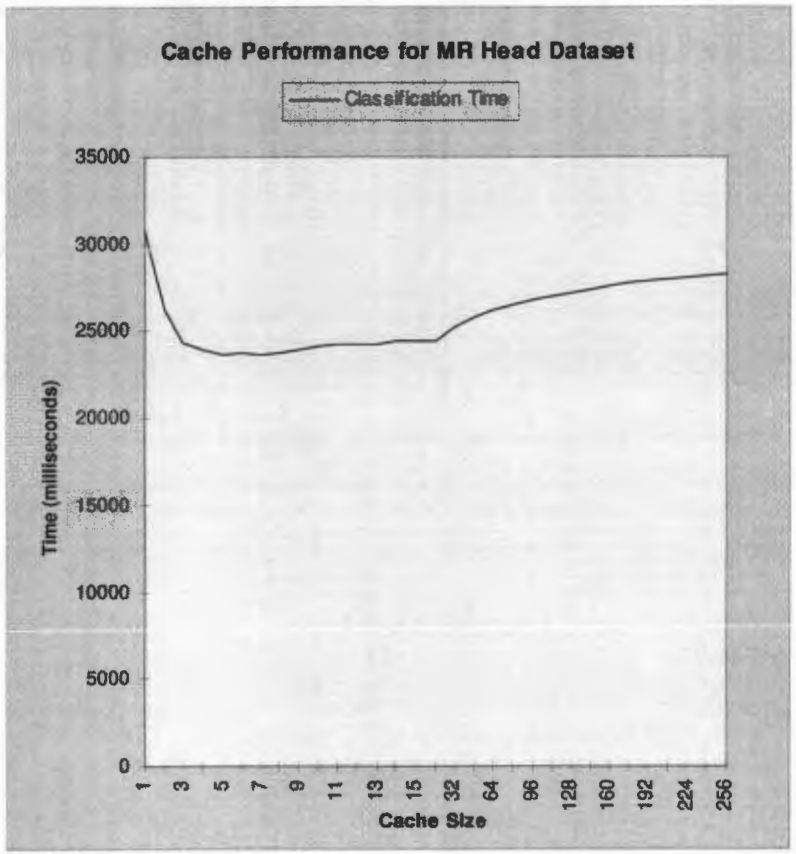


Figure C.24 - Classification performance on the MR Head dataset for varying cache sizes.

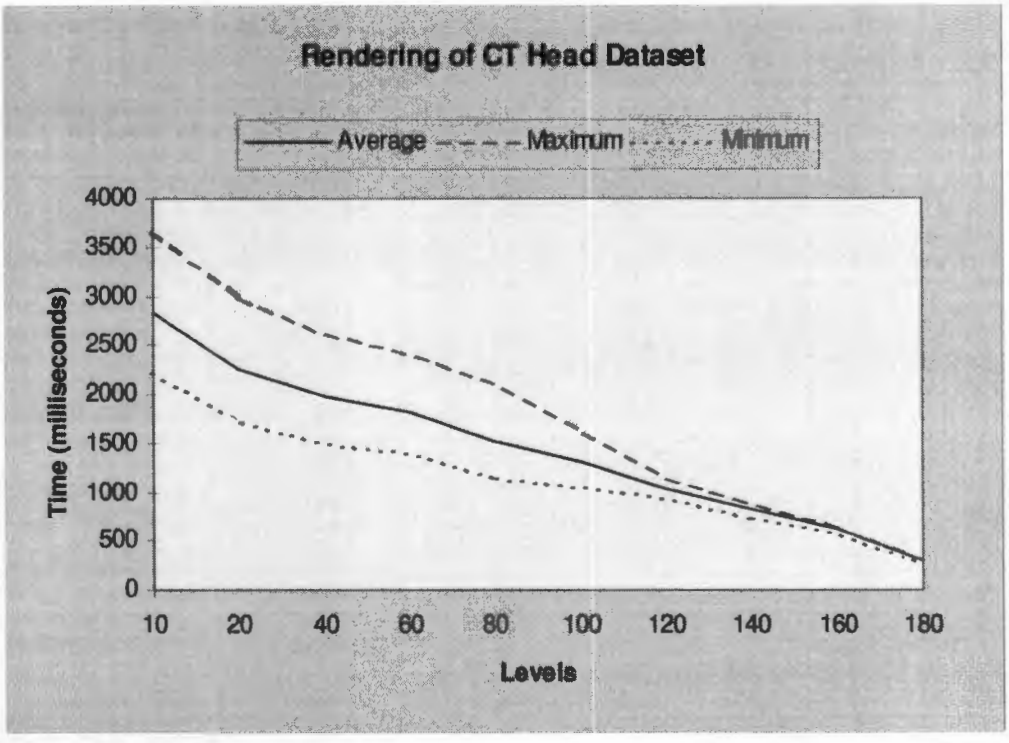


Figure C.25 - Minimum, maximum, and average parallel octree rendering times of the CT Head dataset.

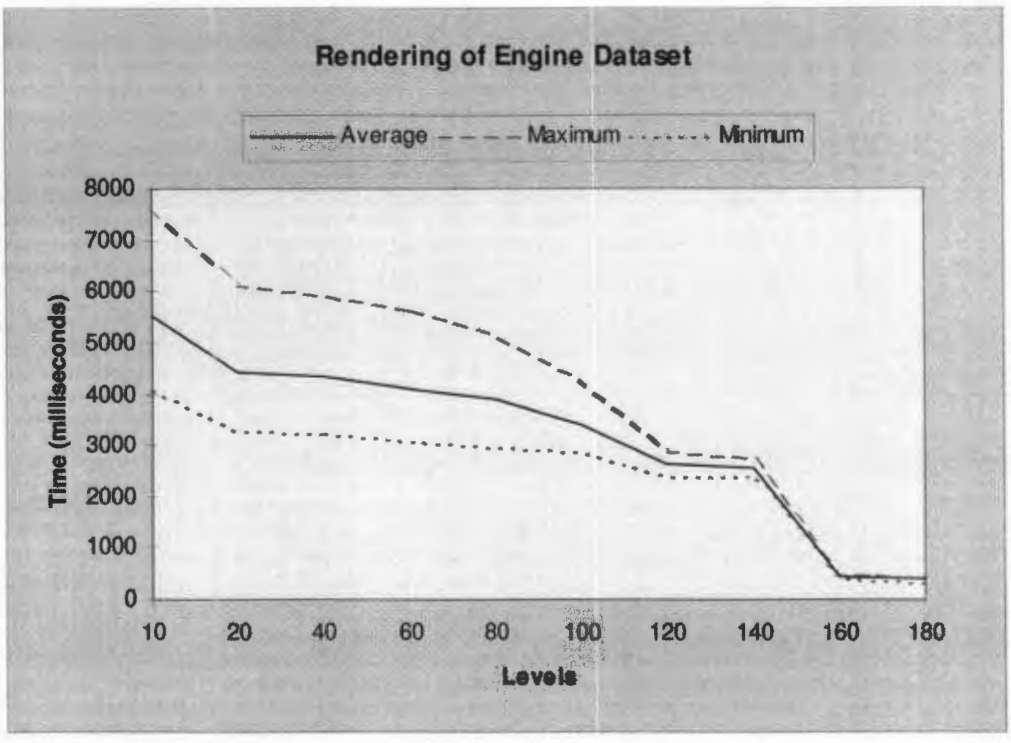


Figure C.26 - Minimum, maximum, and average parallel octree rendering times of the Engine dataset.

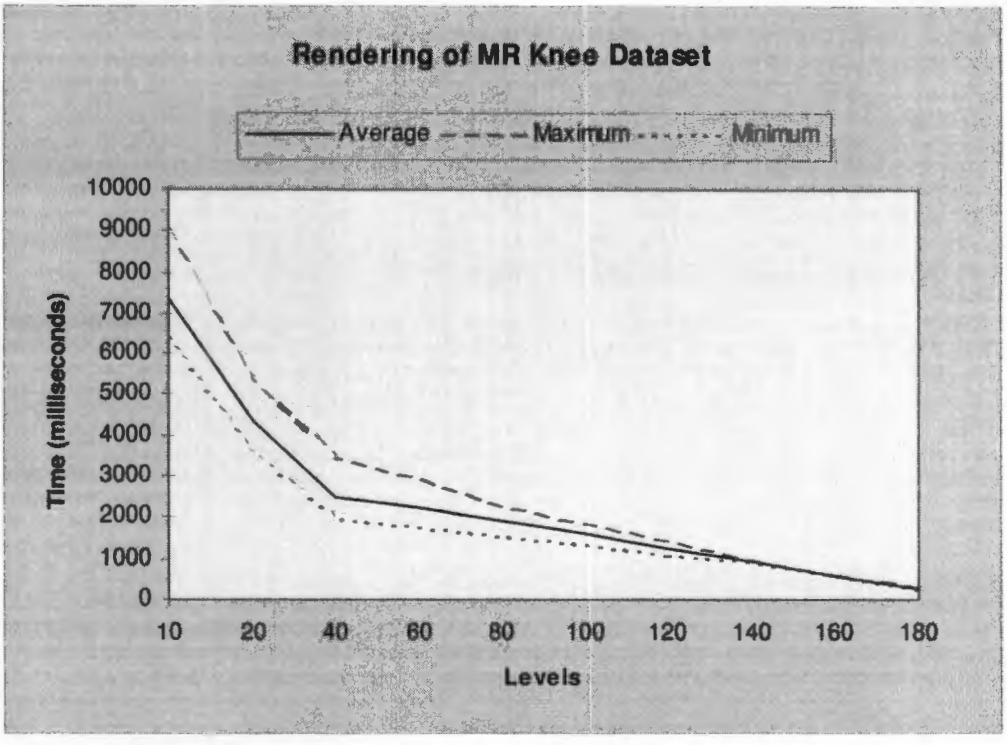


Figure C.27 - Minimum, maximum, and average parallel octree rendering times of the MR Knee dataset.

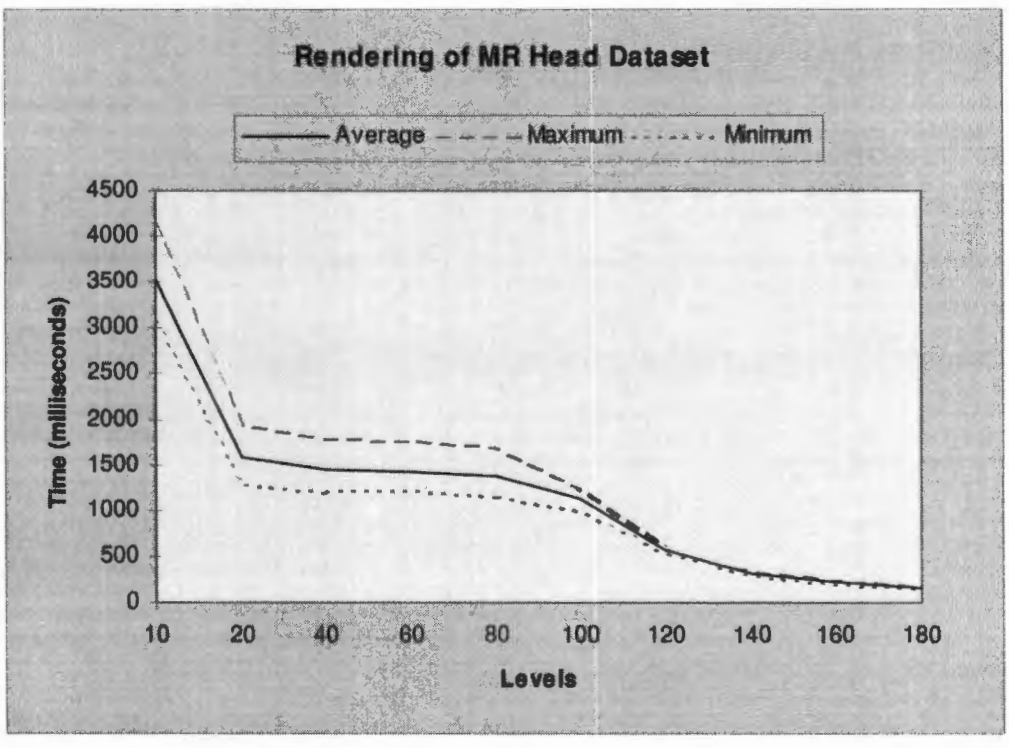


Figure C.28 - Minimum, maximum, and average parallel octree rendering times of the MR Head dataset.

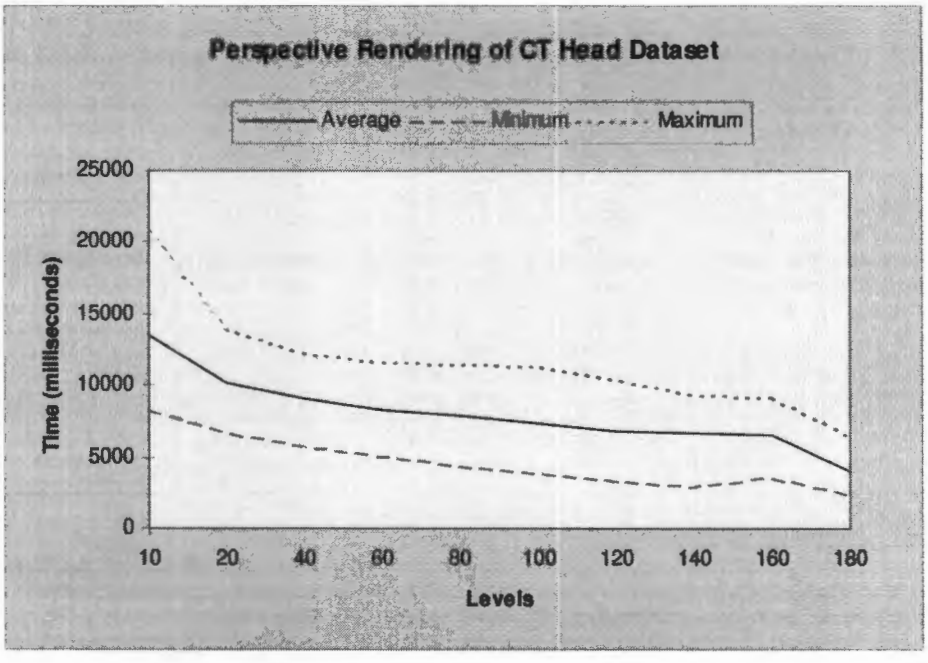


Figure C.29 - Minimum, maximum, and average perspective octree rendering times of the CT Head dataset.

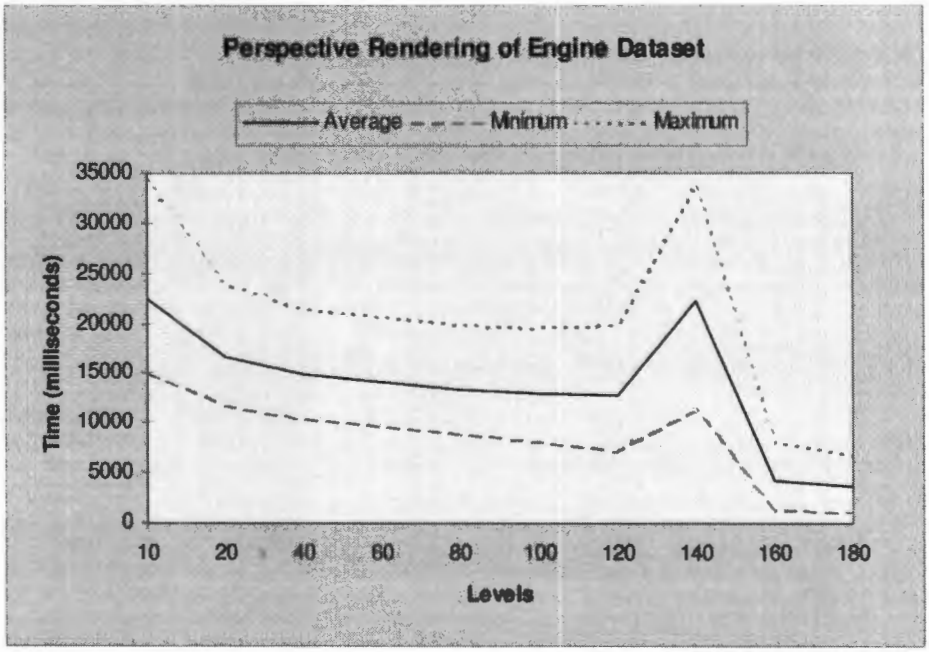


Figure C.30 - Minimum, maximum, and average perspective octree rendering times of the Engine dataset.

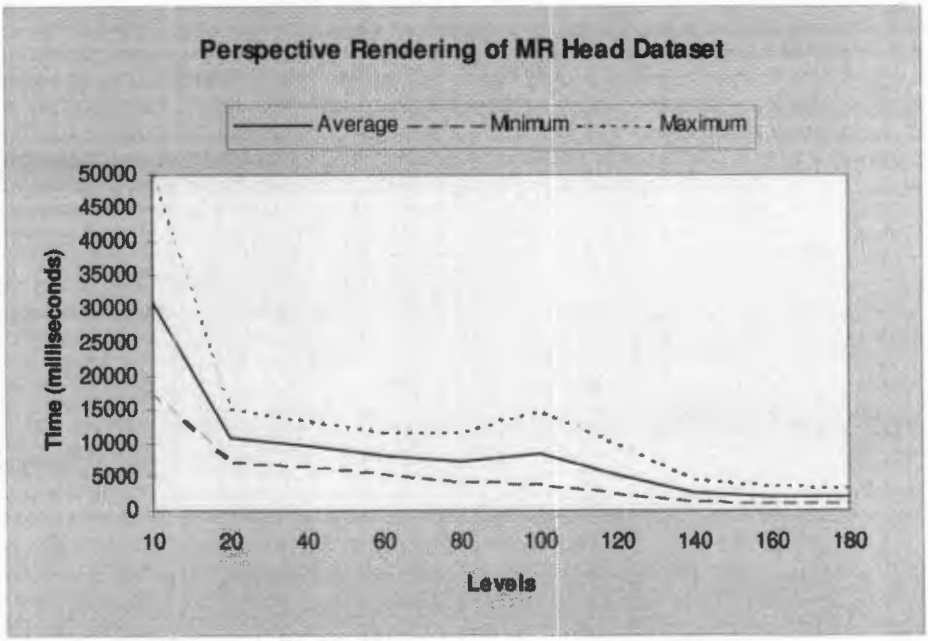


Figure C.31 - Minimum, maximum, and average perspective octree rendering times of the MR Head dataset.

Appendix D

Images

D.1 Standard Parallel Renderings

Figure D.1 - RLE rendering of the CT Head dataset using viewpoint 1.

Figure D.2 - Octree rendering of the CT Head dataset using viewpoint 1. Aliasing bands can be noticed along the top of the skull.

Figure D.3 - RLE rendering of the CT Head dataset using viewpoint 2.

Figure D.4 - Octree rendering of the CT Head dataset using viewpoint 2.

Figure D.5 - RLE rendering of the Engine dataset using viewpoint 1. Interesting aliasing artifacts can be seen on the "rough" side of the volume.

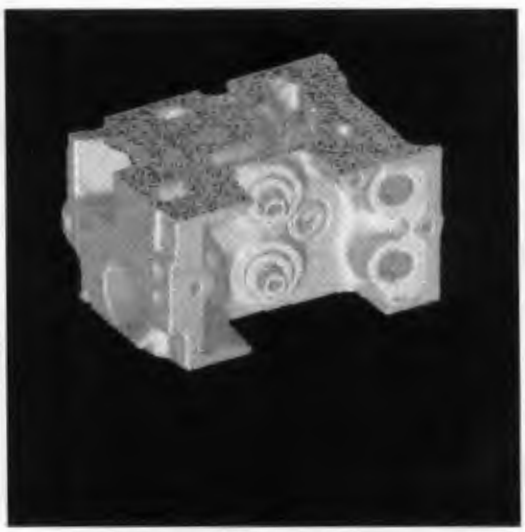


Figure D.6 - Octree rendering of the Engine dataset using viewpoint 1. Very few aliasing artifacts can be noticed on this volume. Octree rendering responds well to Euclidean shapes.

Figure D.7 - RLE rendering of the Engine dataset using viewpoint 2.

Figure D.8 - Octree rendering of the Engine dataset using viewpoint 2.

Figure D.9 - RLE rendering of the MR Head dataset using viewpoint 1.

Figure D.10 - Octree rendering of the MR Head dataset using viewpoint 1.

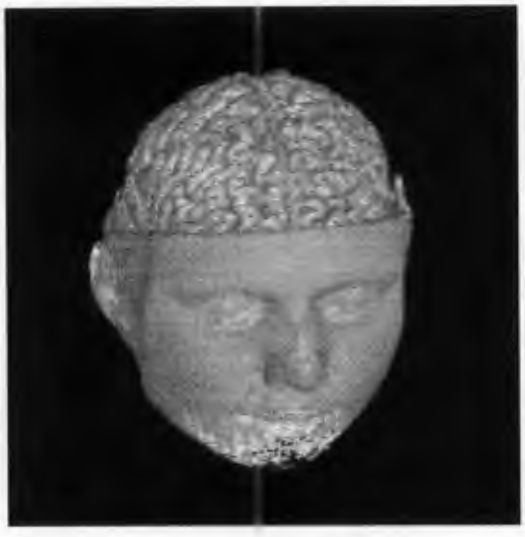


Figure D.11 - RLE rendering of the MR Head dataset using viewpoint 2.

Figure D.12 - Octree rendering of the MR Head dataset using viewpoint 2.

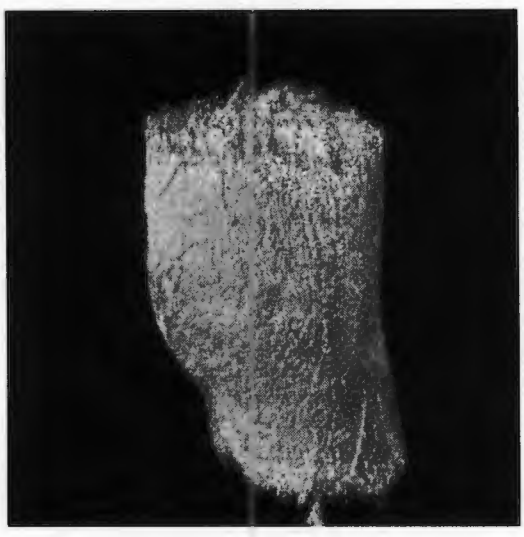


Figure D.13 - RLE rendering of the MR Knee dataset using viewpoint 1.

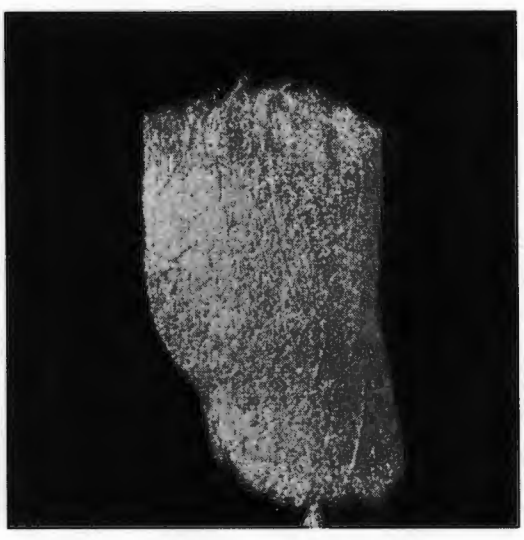


Figure D.14 - Octree rendering of the MR Knee dataset using viewpoint 1.

Figure D.15 - RLE rendering of the MR Knee dataset using viewpoint 2.

Figure D.16 - Octree rendering of the MR Knee dataset using viewpoint 2.

D.2 Perspective Renderings

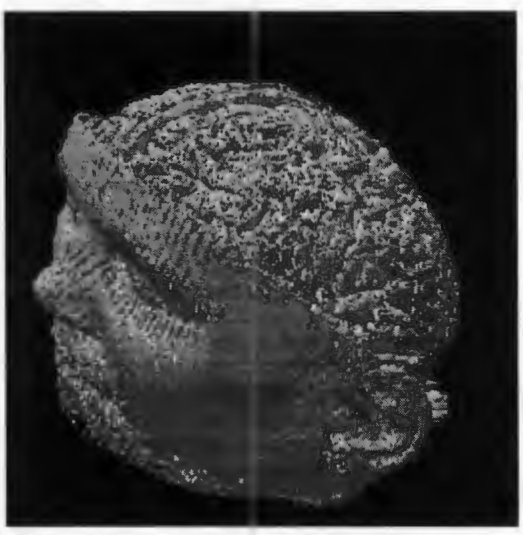


Figure D.17 - Perspective octree rendering of the MR Head dataset.

Figure D.18 - Partial perspective octree rendering of the MR Head dataset. All leaf-nodes are approximated.

Figure D.19 - Perspective octree rendering of the Engine dataset.

Figure D.20 - Partial perspective octree rendering of the Engine dataset. All leaf-nodes are approximated.

Figure D.21 - Perspective octree rendering of the CT Head dataset. The severe aliasing bands are a result of low quality image warping.

Figure D.22 - Partial perspective octree rendering of the CT Head dataset. All leaf-nodes are approximated.

D.3 Using Translucency

Figure D.23 - Parallel RLE rendering of the Engine dataset using viewpoint 1. The outer shell of the engine is set to translucent blue.

Figure D.24 - Parallel octree rendering of the Engine dataset using viewpoint 1. The outer shell of the engine is set to translucent blue.

Figure D.25 - Parallel RLE rendering of the Engine dataset using viewpoint 2. The outer shell of the engine is set to translucent blue.

Figure D.26 - Parallel octree rendering of the Engine dataset using viewpoint 2. The outer shell of the engine is set to translucent blue.

D.4 Partial Parallel Renderings

Figure D.27 - Partial parallel rendering of the CT Head dataset using viewpoint 1. All leaf-nodes of the octree are approximated.

Figure D.28 - Partial parallel rendering of the CT Head dataset using viewpoint 2. All leaf-nodes of the octree are approximated.

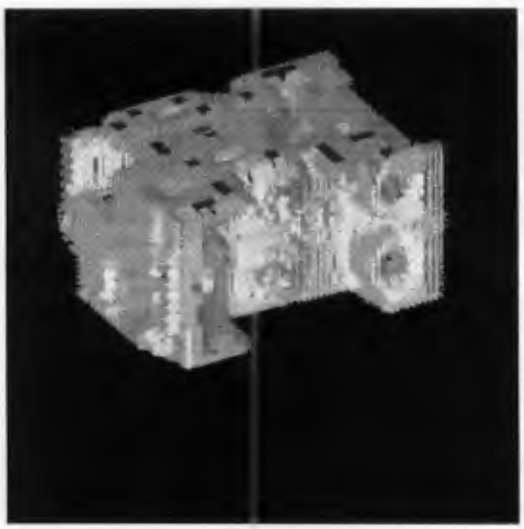


Figure D.29 - Partial parallel rendering of the Engine dataset using viewpoint 1. All leaf-nodes of the octree are approximated.

Figure D.30 - Partial parallel rendering of the Engine dataset using viewpoint 2. All leaf-nodes of the octree are approximated.

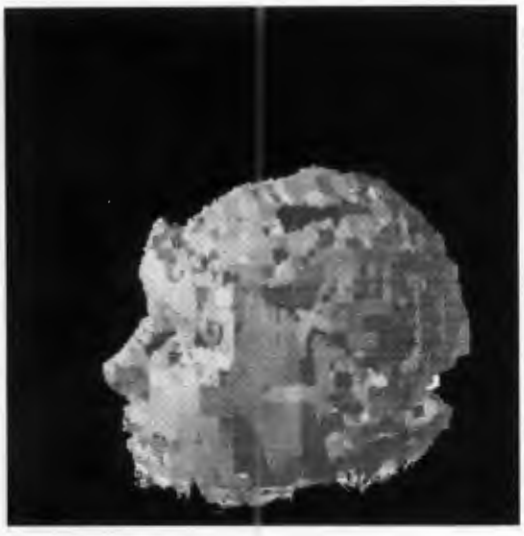


Figure D.31 - Partial parallel rendering of the MR Head dataset using viewpoint 1. All leaf-nodes of the octree are approximated.

Figure D.32 - Partial parallel rendering of the MR Head dataset using viewpoint 2. All leaf-nodes of the octree are approximated.

Figure D.33 - Partial parallel rendering of the MR Knee dataset using viewpoint 1. All leaf-nodes of the octree are approximated

Figure D.34 - Partial parallel rendering of the MR Knee dataset using viewpoint 1. All leaf-nodes of the octree are approximated

Figure D.35 - Partial parallel rendering of the MR Head dataset using viewpoint 1. Data size used is 65536 bytes.

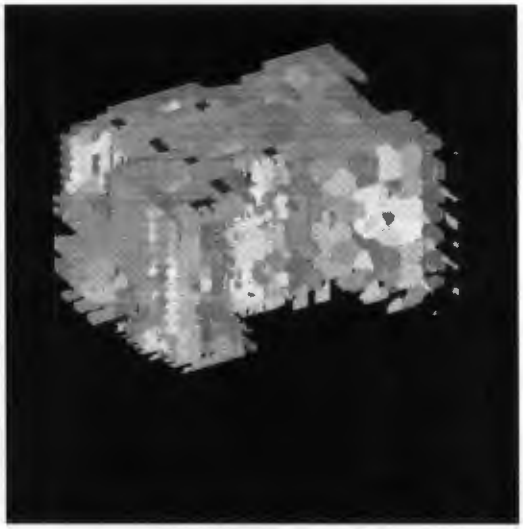


Figure D.36 - Partial parallel rendering of the Engine dataset using viewpoint 1. Data size used is 65536 bytes.

Figure D.37 - Partial parallel rendering of the MR Head dataset using viewpoint 1. Data size used is 131072 bytes.

Figure D.38 - Partial parallel rendering of the Engine dataset using viewpoint 1. Data size used is 131072 bytes.

Figure D.39 - Partial parallel rendering of the MR Head dataset using viewpoint 1. Data size used is 262144 bytes.

Figure D.40 - Partial parallel rendering of the Engine dataset using viewpoint 1. Data size used is 262144 bytes.

Figure D.41 - Partial parallel rendering of the MR Head dataset using viewpoint 1. Data size used is 786432 bytes. The top half of the head is no longer being approximated.

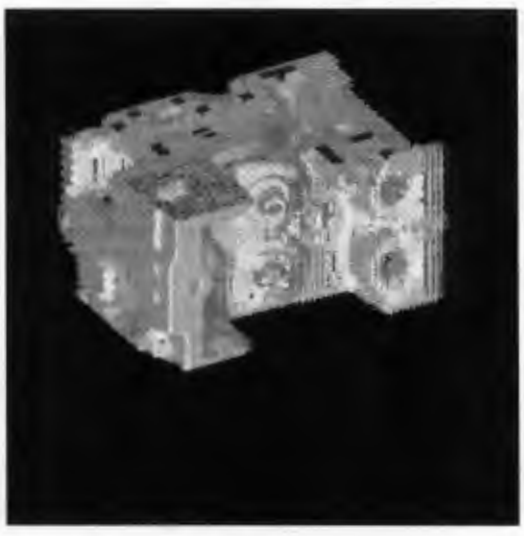


Figure D.42 - Partial parallel rendering of the Engine dataset using viewpoint 1. Data size used is 786432 bytes. Some areas in the front of the engine are no longer being approximated.

Bibliography

[1] T. Lewis. *The Next $10,000_2$ Years: Part I.* IEEE Computer, 29(4):64-70, 1996.

[2] R. Robb, D. Hanson, J. Camp. *Computer-Aided Surgery Planning and Rehearsal at Mayo Clinic.* IEEE Computer, 29(1):39-47, 1996.

[3] G. Bell, A. Parisi, and M. Pesce. *VRML 1.0 Specification.* Available at "http://www.wired.com/vrml.tech/vrml10-3.html".

[4] V. Anupam, C. Bajaj, D. Schikore, M. Schikore. *Distributed and Collaborative Visualisation.* IEEE Computer, 27(7):37-43, 1994.

[5] A. Law and R. Yagel. *Multi-Frame Thrashless Ray Casting with Advancing Ray-Front.* Proceedings Graphics Interface '96, Toronto, Canada, 70-70, May 1996.

[6] R. Simon, D. Krieger, T. Znati, R. Loflink, R. Sclabassi. *Multimedia Mednet.* IEEE Computer, 28(5):65-73, 1995.

[7] S.A. Cheong, D C. Martin, and M.D. Doyle. *Integrated Control of Distributed Volume Visualisation Through the World-Wide-Web.* IEEE Visualisation '94 Proceedings, 13-20, 1994.

[8] C. Giertsen. *Volume Visualisation of Sparse Irregular Meshes.* IEEE Computer Graphics and Applications, 12(2):40-48, 1992.

[9] M. Garrity. *Raytracing Irregular Volume Data.* ACM Siggraph, San Diego Workshop on Volume Visualization, 24(5):35-40, 1990.

[10] Jane Wilhelms. *Visualizing sampled volume data.* In "Scientific Visualization and Graphics Simulation", chapter 6, Wiley, 1990.

[11] J. Wilhelms and A. Van Gelder. *Octrees for faster isosurface generation, extended abstract.* ACM Siggraph, San Diego Workshop on Volume Visualization, 24(5):57-62, 1990.

[12] D. Laur and P. Hanrahan. *Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering.* ACM Siggraph Computer Graphics Proceedings, 25(4):285-288, 1991.

[13] M. Levoy. *Efficient ray tracing of volume data.* ACM Transactions on Graphics, 9(3):245-261, 1990.

[14] S. Muraki. *Volumetric Shape Description of Range Data using "Blobby Model".* ACM Siggraph Computer Graphics Proceedings, 25(4):227-235, 1991.

[15] E. Stollnitz, T. DeRose, D. Salesin. *Wavelets for Computer Graphics (Part.1).* IEEE Computer Graphics and Applications, 15(3):76-84, 1995.

[16] R. Westermann. *Multiresolution Framework for Volume Rendering.* ACM Siggraph Symposium on Volume Visualization, 51-57, 1994.

[17] S. Muraki. *Volume Data and Wavelet Transforms.* IEEE Computer Graphics and Applications, 13(4):50-56, 1993.

[18] S. Muraki. *Multiscale Volume Representation by a DoG Wavelet.* IEEE Transactions on Visualisation and Computer Graphics, 1(2):109-116, 1995.

[19] B. Guo. *A Multiscale Model for Structure-Based Volume Rendering.* IEEE Transactions on Visualisation and Computer Graphics, 1(4):291-301, 1995.

[20] V. Ranjan and A Fournier. *Volume Models for Volumetric Data.* IEEE Computer, 27(7):28-36, 1994.

[21] J. Wilhelms and A. Van Gelder. *Multi-Dimensional Trees for Controlled Volume Rendering and Compression.* ACM Siggraph Symposium on Volume Visualization, 27-34, 1994.

[22] C. Montani and R. Scopigno. *Rendering volumetric data using the sticks representation scheme.* ACM Siggraph, San Diego Workshop on Volume Visualization, 24(5):87-93, 1990.

[23] N. Shareef, and R. Yagel. *Rapid Previewing via Volume-based Solid Modeling.* The Third Symposium on Solid Modeling and Applications, SOLID MODELING'95, Utah, 281-292, May 1995

[24] P. Lacroute and M. Levoy. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* ACM Siggraph Computer Graphics Proceedings, (Annual Conference Series):451-458, 1994.

[25] P. Lacroute. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation.* PhD. Thesis, Stanford University, Computer Graphics Laboratory, 1995.

[26] J. Fowler and R. Yagel. *Lossless Compression of Volume Data.* ACM Siggraph Symposium on Volume Visualization, 43-50, 1994.

[27] P. Ning and L. Hesselink. *Fast Volume Rendering of Compressed Data.* IEEE Visualisation Proceedings '93, 11-18, 1993.

[28] Y. Boon-Lock and L. Bede. *Volume Rendering of DCT-Based Compressed 3D Scalar Data.* IEEE Transactions on Visualization and Computer Graphics, 1(1):29-43, 1995.

[29] W.E. Lorenson and H.E. Cline. *Marching cubes: A High Resolution 3-D Surface Construction Algorithm.* ACM Siggraph Computer Graphics Proceedings, 24(5):163-169, 1987.

[30] W. Kruger. *Volume Rendering and Data Feature Enhancement.* ACM Computer Graphics, San Diego Workshop on Volume Visualization, 24(5):21-26, 1990.

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

[32] L. Sobierajski, D. Cohen, A. Kaufman, R. Yagel, and D. Acker. *A Fast Display Method for Volumetric Data.* The Visual Computer, 10(2):116-124, 1993.

[33] J.K. Udupa and D. Odhner. *Fast visualisation, manipulation, and analysis of binary volumetric objects.* IEEE Computer Graphics and Applications, 11(6):53-62, 1991.

[34] R. Yagel, D Stredney, G.J. Wiet, P. Schmalbrock, L. Rosenberg, D.J. Sessanna, Y. Kurzion, and S. King. *Multisensory Platform for Surgical Simulation.* IEEE Virtual Reality Annual International Symposium 1996, Santa Clara, California, 72-78, March 1996.

[35] K. Zuiderveld. *Visualisation of Multimodality Medical Volume Data using Object-Oriented Methods.* PhD. Thesis, University of Utrecht, Faculty of Science, 1995.

[36] K.L. Novins, F.X. Sillion, and D.P. Greenberg. *An Efficient method for Volume Rendering using Perspective Projection.* ACM Computer Graphics, San Diego Workshop on Volume Visualization, 24(5):95-100, 1990.

[37] C.T. Howie and E.H. Blake. *The Mesh Propagation Algorithm for Isosurface Construction.* Computer Graphics Forum, Eurographics, 13(3):64-74, 1994.

[38] A. Kaufman, D. Cohen, and R. Yagel. *Volumetic Graphics.* IEEE Computer, 26(7):51-64, July 1993.

[39] T. Totsuka and M. Levoy. *Frequency Domain Volume Rendering.* ACM Siggraph Computer Graphics Proceedings, (Annual Conference Series):271-278, 1993

[40] R.Yagel and A. Kaufman. *Template-Based Volume Viewing.* Computer Graphics Forum, Eurographics, 11(3):153-167, 1992.

[41] L.M. Sobierajski and A.E. Kaufman. *Volumetric Ray Tracing.* ACM Siggraph Symposium on Volume Visualization, 11-18, 1994.

[42] R. Avila, L Sobierajski, and A. Kaufman. *Towards a Comprehensive Volume Visualization System.* Proceedings Visualisation '92, Boston, 13-20, October 1992.

[43] R. Yagel and Z.Shi. *Accelerating Volume Animation by Space Leaping.* Proceedings of Visualisation '93, San Jose, California, 62-69, October 1993.

[44] K.L. Novins, F.X. Sillion, and D.P. Greenberg. *An Efficient method for Volume Rendering using Perspective Projection.* ACM Computer Graphics, San Diego Workshop on Volume Visualization, 24(5):95-100, 1990.

[45] J. Wilhelms and A. Van Gelder. *A Coherent Projection Approach for Direct Volume Rendering.* ACM Siggraph Computer Graphics Proceedings, 25(4):275-284, 1991.

[46] T. Porter and T. Duff. *Compositing Digital Images.* ACM Computer Graphics, 18(3), 1984.

[47] B. Cabral, N. Cam, J. Foran. *Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware.* ACM Siggraph Symposium on Volume Visualization, 91-97, 1994.

[48] T.J. Cullip, and U. Neumann. *Accelerating Volume Reconstruction With 3D Texture Hardware.* Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill, 1993.

[49] L. Lippert and M. Gross. *Fast Wavelet Based Volume Rendering by Accumulation of Transparent Texture Maps.* Computer Graphics Forum, Eurographics, 14(3):431-443, 1995.

[50] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping.* MSc. Thesis, University of California (Berkeley), Department of Electrical Engineering and Computer Science, 1989.

[51] G. Wolberg. *Digital Image Warping.* IEEE Computer Society Press, 1994, Third Edition.

[52] P. Hanrahan. *Three-Pass Affine Transforms for Volume Rendering.* ACM Siggraph, San Diego Workshop on Volume Visualization, 24(5):71-77, 1990.

[53] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[54] C.A. Shaffer. *Bit Interleaving for Quad- or Octrees.* In "Graphics Gems I", 443-447, Academic Press 1990.

[55] D.P. Anderson. *Hidden Line Elimination in Projected Grid Surfaces.* ACM Transactions on Graphics, 1(4): 274-288, 1982.

[56] J. Foley, A. van Dam, S. Feiner, J. Hughes. *Computer Graphics - Principles and Practice.* Addison-Wesley Publishing Company, 1991, Second Edition.

[57] M. Haley and E. Blake. *Incremental Volume Rendering Using Hierarchical Compression.* Computer Graphics Forum, Eurographics. (1996) ... *Accepted for publication.*