Web-based Rendering Techniques for Large-scale Biomedical Data Sets

Joerg Meyer
NSF-Engineering Research Center (ERC), Mississippi State University
2 Research Blvd., Starkville, MS 39762-9627
jmeyer@cs.msstate.edu

Ragnar Borg, Bernd Hamann, Kenneth I. Joy
Center for Image Processing and Integrated Computing (CIPIC)
Department of Computer Science - University of California - One Shields Ave.
Davis, CA 95616-8562
Ragnar.Borg@proxycom.no, hamann@cs.ucdavis.edu, joy@cs.ucdavis.edu

Arthur J. Olson
The Scripps Research Institute
10550 North Torrey Pines Road - La Jolla, CA 92037
olson@scripps.edu

Abstract

Large volumetric data sets are usually stored as file sets, where each individual file represents an orthogonal cross-section. Interactive rendering of large data sets requires fast access to user-defined parts of the data, because it is virtually impossible to render the entire data set at full resolution. Therefore, hierarchical rendering techniques have been introduced to render a region-of-interest at a higher resolution than the rest of the data. Lower levels of detail are provided as context information. We present a dynamic subdivision scheme, which incorporates space subdivision and wavelet compression.

1 Introduction

Real-color volume data sets can be obtained by taking photographs or scanning cross-sections of objects. These objects are typically in a frozen state (cryo-sections). This technique produces high resolution image data in real-color. The resolution is only limited by the camera or the imaging device, and not so much by principal limitations of the scanning device, because there is no complex matrix transformation required to obtain 2D image data, as it is necessary for CT or MRI.

Therefore, real-color volume data sets tend to be much more voluminous than CT or MRI imagery.

In order to make a data set available on a visualization server and transmit the data progressively to a rendering client, we need to compactify the data set and break it down into smaller bricks. The order of transmission and the size or resolution of the bricks is determined and driven by the client application. Our system uses a Windows NT-based server system which is both data repository and content provider for shared rendering applications. The client accesses the server via a web-based interface.

The client selects a data set and sends a request to the server. The server analyzes and interprets the request and returns a customized Java applet together with an appropriate representation of the data set. The Java applet is optimized for a specific rendering task. This means that the rendering algorithm is tailored for a particular problem set. This keeps the applet small and avoids additional overhead and testing for different cases. The initial data set is also small. It is refined later upon additional requests by the client. Bricks of different sizes and different resolutions might be requested from the server. We present a method that combines dynamic space subdivision algorithms, such as adaptive octrees for volumes, wavelet-based data representation, and progressive data transmission for hierarchically stored volume data sets.

2 Indexing scheme

Original data are usually structured as sets of files, which represent a series of 2-D cross-sections. By arranging all slices in a linear array, we obtain a 3-D volume. Unfortunately, when accessing the data, in most cases we do not make use of the implicit coherency across single slices. This coherency is only useful for extraction of cross-sections perpendicular to the scanning direction, i. e., within a single image plane. Instead, in most cases we need brick-like coherency within subvolumes. Therefore, we present a new datastructure, which uses a combination of delimited octree space subdivision and wavelet compression techniques to achieve better performance.

We present an efficient indexing scheme, an adaptive data reduction method, and an efficient compression scheme. All techniques are based on integer arithmetic and are optimized for speed. Binary bit operations allow for memory efficient storage and access.

We use a standard filesystem (Unix or FAT32) to store our derived datastructures, and we use filenames as keys to the database. This way we can avoid additional overhead, which is typically caused by inserting additional access layers between the application and the underlying storage system. We found that this method provides the fastest method to access the data. Our indexing scheme in conjunction with the underlying filesystem provides the database system (repository) for the server application, which reads the data from the repository and sends it to a remote rendering client upon request. Initially, a low resolution representation is requested from the repository and rendered on the client side. This coarse representation provides context cues and sufficient information for initial navigation. After the user has specified a subvolume or region-of-interest (ROI), the client application sends a new request to the server to retrieve a subvolume at a higher level of detail. When using the data structures described below, this updating procedure typically takes considerably less time compared to the single-slice representation, because a smaller number of files needs to be touched. The initial step, which requires reading the initial section of every file, i.e., all bricks, can be sped up by storing an additional file which contains a reduced version of the entire data set.

3 Storage scheme

The filesize *f* for storing the leaves of the octree structure should be a multiple *n* of the minimum page size *p* of the filesystem. *p* is typically defined as a system constant in `/usr/include/sys/param.h`). *n* depends on the wavelet compression, which is described below. If the lowest resolution of the subvolume requires *b* bytes, the next level requires a total of $8 \cdot b$ bytes (worst case, uncompressed), and so forth.

We assume that we have a recursion depth *r* for the wavelet representation. This gives us $8^r \cdot b$ bytes, which must fit in *f*. This means:

$$f = n \cdot p \geq 8^r \cdot b \tag{1}$$

Both *r* and *b* are user-defined constants. Typical values are $b = 512$, which corresponds to an $8 \times 8 \times 8$ subvolume, and $r = 3$, which gives us four levels of detail over a range between $8^0 \cdot 512 = 512$ and $8^3 \cdot 512 = 262144$ data elements, which is more than 2.7 orders of magnitude.

For optimal performance and in order to avoid gaps in the allocated files, we can assume that

$$n \cdot p = 8^r \cdot b, \tag{2}$$

thus

$$n = 8^r \cdot \frac{b}{p}. \tag{3}$$

The enormous size of the data sets (see chapter 4) requires that the data is subdivided into smaller chunks, which can be loaded into core memory within a reasonable amount of time [1, 5]. Since we are extracting subvolumes, it seems quite natural to break the data up into smaller bricks. This can be done recursively using an octree method [3, 4, 7]. Each octant is subdivided until we reach an empty region which does not need to be subdivided any further, or until we hit the filesize limit *f*, which means that the current leaf fits into a file of the given size.

Each leaf contains the full resolution. Memory space is reduced by skipping empty regions. Typically, the size of the data set shrinks to about 20%, i.e., one fifth of the original size (see chapter 5).

Since we want to access the data set in a hierarchical fashion, we have to convert the leaves into a multiresolution representation. This representation must be chosen in a way that the reconstruction can be performed most efficiently with minimal computational effort [8]. Haar wavelets fulfill these properties. They also have the advantage that they can be easily implemented as integer arithmetic. The lowest resolution (figure 1, lower right) is stored at the beginning of the file, thus avoiding long seek times within the file.

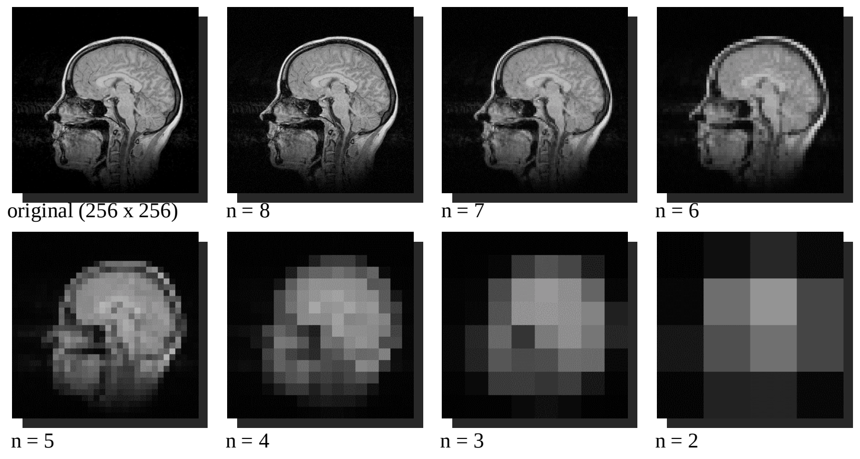


Figure 1. Haar wavelet compression scheme (2-D)

For the wavelet representation, we associate each subvolume with a vector space *V*, which consists of a set of piecewise linear functions. V^0 is associated with a constant function, which is defined over the domain [0, 1[, and describes a single pixel. V^i consists of 2^i intervals, with a constant function defined on each of these intervals [6]. All vector spaces are subsets of each other:

$$V^i \subset V^{i+1}, i \in \mathbb{N}_0 \tag{4}$$

We choose the following basis functions for V^i:

$$\phi_{i,j}(x) = \phi(2^i x - j), j \in \{0, \ldots, 2^i - 1\} \quad \text{with}$$
$$\phi(x) = \begin{cases} 1 & \text{if } 0 \le x < 1 \\ 0 & \text{else} \end{cases} \tag{5}$$

Figure 2 shows the basis functions, also called scaling functions.

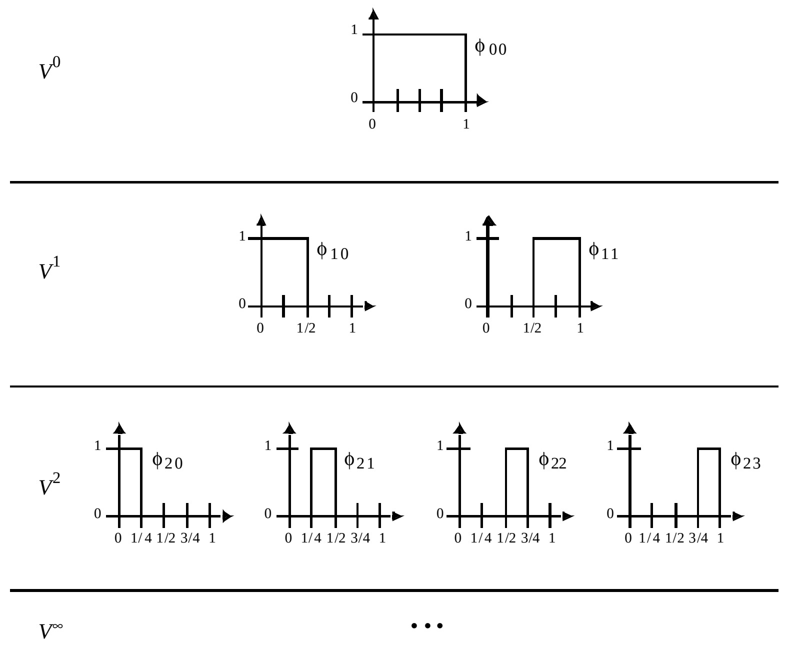


Figure 2. Wavelet transformation: scaling functions

We define another vector space W^i, which comprises all functions of V^{i+1}, and which is orthogonal to all functions in V^i. These basis functions, which span W^i, are the *Haar wavelets*:

$$\psi_{i,j}(x) = \psi(2^i x - j),\ j \in \{0,\ldots,2^i - 1\} \quad \text{with}$$

$$\psi(x) = \begin{cases} 1 & \text{if } 0 \le x < \frac{1}{2} \\ -1 & \text{if } \frac{1}{2} \le x < 1 \\ 0 & \text{else} \end{cases} \tag{6}$$

Figure 3 shows these basis functions.

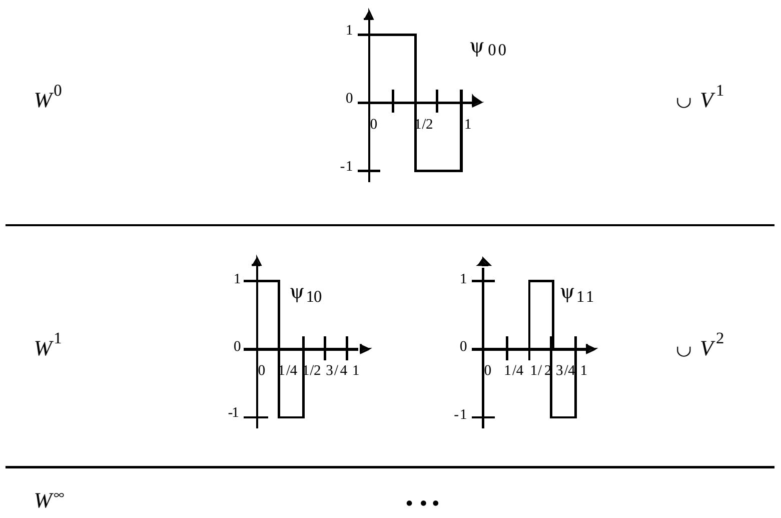


Figure 3. Haar wavelet basis functions

A discrete signal in V^{i+1} can be represented as a linear combination of the basis functions from V^0 and W^0 ... W^i.

We can describe the original image I as follows (pixel data: $l_i, i \in \mathbb{N}_0$):

$$I(x) = \sum_{i=1}^{n} l_i \cdot \phi_{\frac{n}{2}-1,i-1}(x) \tag{7}$$

After the first transformation, we obtain:

$$I(x) = \sum_{i=1}^{n/2} l'_i \cdot \phi_{\frac{n}{2}-2,i-1}(x) + \sum_{i=1}^{n/2} c'_i \cdot \psi_{\frac{n}{2}-2,i-1}(x) \tag{8}$$

where the first sum represents the reduced image, while the second sum comprises the detail coefficients.

Now we lift this scheme to the three-dimensional case. A simple solution would be an enumeration of all grid points, e. g., row by row and slice by slice, in a linear chain, so that we can still apply our one-dimensional algorithm (standard decomposition). This of course would reduce the data set only by a factor of 2, instead of 2^3 = 8. Thus we apply the algorithm alternatively to each dimension. Figure 4 shows the method for the two-dimensional case [8].

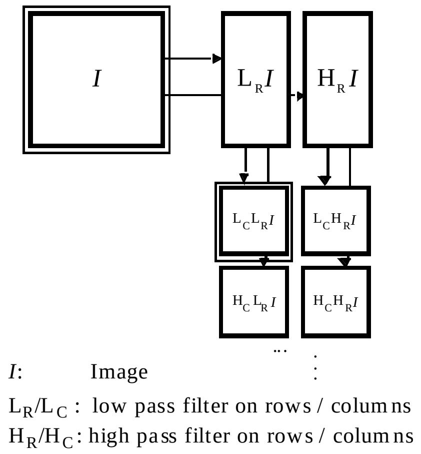

I
L_R I
H_R I
L_C L_R I
L_C H_R I
H_C L_R I
H_C H_R I
I: Image
L_R/L_C : low pass filter on rows / columns
H_R/H_C: high pass filter on rows / columns

Figure 4. Wavelet compression scheme

First, all rows of the original image *I* are decomposed into a low-pass filtered image $L_R I$ (reduced image) and the high-pass filtered components $H_R I$ (detail coefficients). For the next transformation, the algorithm is applied to the columns, which results in $L_C L_R I$, $H_C L_R I$, $L_C H_R I$, and $H_C H_R I$ (prefix notation). The same technique can be used for three dimensions.

After each cycle, we end up with a reduced image in the upper left corner. Subsequently, the algorithm is only applied to this quadrant (see figure 5). The algorithm terminates if the size of this quadrant is one pixel in each dimension.

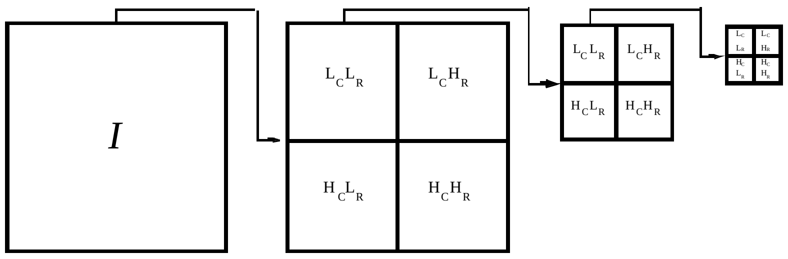


Figure 5. Wavelet compression: memory management

This method of alternating between dimensions is known as non-standard decomposition. Figure 1 shows the $L_C L_R$ components for an MRI scan at different levels of detail.

A very useful property is the fact that even for lossless compression a volume converted into the wavelet representation requires the exact same amount of memory as the original representation. Since many coeeficients are relatively small, the number of different discrete values is also small, provided we use integer arithmetic. Extremely small values can be neglected in order to obtain better compression rates (lossy compression). We use a simple run-length encoding (RLE) scheme, which turns out to be efficient, especially for small brick sizes *b*, and it allows for easy decoding. The space requirement (lossless compression) is the same for all subsequent wavelet recursions, i. e., for all levels of detail. The wavelet algorithm terminates when it reaches a predifined minimum subvolume size *b*. The lower bound is the size of a single voxel.

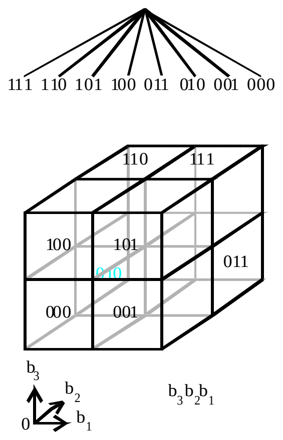


Figure 6. Numbering scheme

Each octant can be described by a number [2]. We use the following numbering scheme (figure 6): A leaf is uniquely characterized by the octree recusion depth and the octree path. We limit the recursion depth to eight, which allows us to encode the depth in 3 bits. In order to store the path, we need 3 bits per recursion step, which gives us 24 bits. 4 bits are spent to encode the depth of the wavelet recursion. The remaining bit is a flag which indicates if the file is empty or not. This prevents us from opening and attempting to read the file and speeds up the computation. The total number of bits is 32 (double word).

3	3	. . .	3	4	1
oct.depth	sub 1		sub 8	wav.depth	empty

Figure 7. Tree encoding

Each bit group can be easily converted into an ASCII character by using binary arithmetic, e.g., `(OCT_DEPTH >> 29) | 0x30)` would encode the octree depth as an ASCII digit. By appending these characters we can generate a unique filename for each leaf.

In order to retrieve a subvolume, we have to find the file(s) in which the corresponding parts are stored. We start with the lower left front corner and identify the subvoxel by recursive binary subdivision of the bounding box for each direction. Each decision gives us one bit of the subvolume path information. We convert these bits into ASCII characters, using the same macros as above. The first file we are looking for is `7xxxxxxxx??`, where the ’`x`’s describe the path, and ’`?`’ is a wildcard. If this file does not exist, we keep looking for `6xxxxxxx???`, and so forth, until we find an existing leaf. If the filename indicates that the file is empty (last digit), we can skip the file. The filename also indicates how many levels of detail we have available for a particular leaf. This allows us to scale the rendering algorithm. In order to retrieve the rest of the subvolume, we must repeat this procedure for the neighboring leaves. The number of iterations depends on the recusion depth and therefore on the size of the leaves found. The algorithm terminates when all files have been retrieved so that the subvolume is complete.

4 Results

Our test application focusses on biomedical imaging. A prototype was designed to support three-dimensional visualization of a human brain, which allows us to study details by moving tools, such as an arbitrary cutting plane and variously shaped lenses, across the data set. The various data sets are typically between 20 MB and 76 GB, which makes them impossible to transfer over the internet in real time. The rendering client operates independently from the size of the data set and requests only as much data as can be displayed and handled by the Java applet.

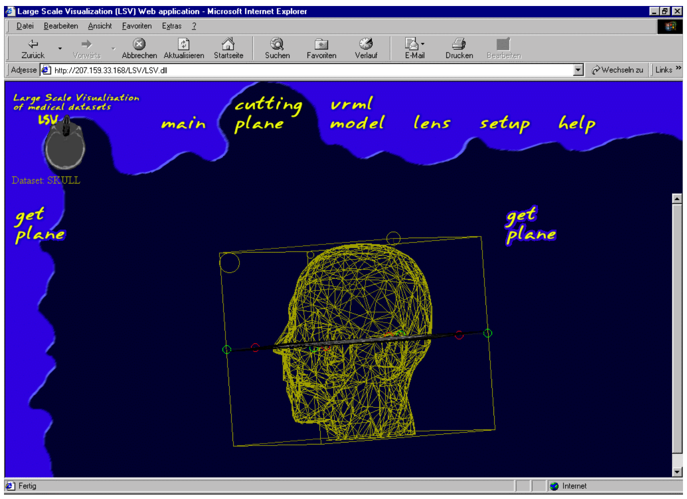

Large Scale Visualization (LSV) Web application - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Vorwärts Abbrechen Aktualisieren Startseite Suchen Favoriten Verlauf E-Mail Drucken Bearbeiten
Adresse http://207.159.33.168/LSV/LSV.dll
Wechseln zu Links
Large Scale Visualization of medical datasets
LSV
main
cutting plane
vrml model
lens
setup
help
Dataset: SKULL
get plane
get plane
Fertig
Internet

Figure 8. Prototype implementation

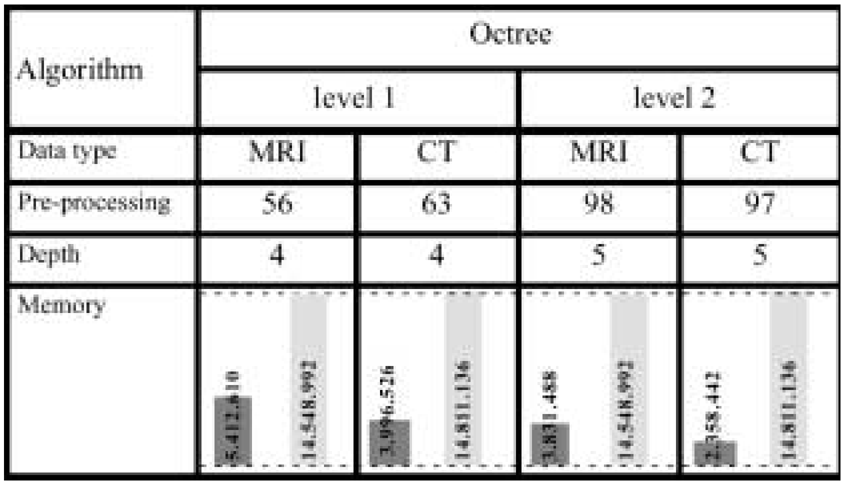
The web-based user interface combines HTML-form-driven server requests with customized Java applets, which are transmitted by the server to accomplish a particular rendering task.

Our prototype implementation (figure 8) features 2-D/3-D preview capability; interactive cutting planes (in a 3-D rendering, with hierarchical isosurface models to provide context information); a lens paradigm to examine a particular region-of-interest (variable magnification and lens shape, interactively modifiable ROI); etc. Complex scenes can be precomputed on the server side and transmitted as a VRML2 file to the client so that the client can render the scene, and the user can interact with it in real-time.

5 Comparison

Our new data structure uses considerably less memory than the original data set, even if we choose lossless compression. By selecting appropriate thresholds for the wavelet compression algorithm, we can switch between lossless compression and extremely high compression rates. Computing time is balanced by choosing an appropriate filesize (chapter 3).

One of the advantages of this approach is the fact that the computing time does not so much depend on the resolution of the subvolume, but merely on the size of the subvolume, i. e., the region-of-interest. This is because the higher resolution versions (detail coefficients in conjunction with the lower resolution versions) can be retrieved in almost the same time from disk as the lower resolution version alone. All levels of detail are stored in the same file, and the content of several files, which make up the subvolume, usually fits into main memory. Since seek time is much higher than read time for conventional harddisks, the total time for data retrieval mainly depends on the size of the subvolume, i. e., the number of files that need to be accessed, and not so much on the level of detail.



Algorithm	Octree			
	level 1		level 2	
Data type	MRI	CT	MRI	CT
Pre-processing	56	63	98	97
Depth	4	4	5	5
Memory	5,412,610 / 14,548,992	3,096,526 / 14,811,136	3,831,488 / 14,548,992	2,358,442 / 14,811,136

Table 1. Space subdivision algorithm

Table 1 shows the reduction in the amount of memory required to store a large data set, if we use an octree at two different levels. The column on the right represents the original data set. The wavelet decomposition takes about 0.07 sec for a 64^3 data set, and 68 sec for a 1024^3 data set. The reconstruction can be done more efficiently and usually takes about 30% of the time (measurements based on an R12000 processor). For the above data we assume lossless wavelet decomposition. RLE or other (lossy) compression/decompression algorithms take an additional amount of time, but this is negligible compared to data transmission time.

6 Conclusions

We have presented an efficient numbering scheme and access method for hierarchical storage of subvolumes on a regular filesystem. This method allows us to access a region-of-interest as a set of bricks at various lvels of detail. The simplicity of the method makes it easy to implement. The algorithm is scalable by increasing word length and filename length. Future work includes better wavelet compression schemes, lossy compression techniques, and time-variant data sets.

We are currently working on the integration, adaptation and evaluation of these tools in the National Partnership for Advanced Computational Infrastructure (NPACI) framework. Future research efforts include integration of San Diego Supercomputer Center's High-performance Storage System (HPSS) as a data repository to retrieve large-scale data sets, and accessing the data via NPACI's Scalable Visualization Toolkits (also known as VisTools).

Acknowledgements

Data sets courtesy of Arthur W. Toga, UCLA School of Medicine. Thanks to Bernd Hamann, Kenneth I. Joy, Kwan-Liu Ma, Ikuko Takanashi, Ragnar Borg, Eric B. Lum, UC Davis; Arthur J. Olson, The Scripps Institute; and Edward G. Jones, Neuroscience Center, UC Davis.

References

1. Heiming, Carsten, “Raumunterteilung von Volumendaten,” thesis, Department of Computer Science, University of Kaiserslautern, Germany, January 1998.
2. Hunter, G. M.; Steiglitz, K., “Operations on Images Using Quad Trees,” IEEE Trans. Pattern Anal. Mach. Intell., 1(2), April 1979, 145–154.
3. Jackins, C.; Tanimoto, S. L., “Oct-Trees and Their Use in Representing Three-Dimensional Objects,” CGIP, 14(3), November 1980, 249–270.
4. Meagher, D., “Octree Encoding: A New Technique for the Representation, Manipulation, and Display of Arbitrary 3-D Objects by Computer,” Technical Report IPL-TR-80-111, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, NY, October 1980.
5. Meyer, Joerg; Gelder, Steffen; Heiming, Carsten; Hagen, Hans, “Interactive Rendering—A Time-Based Approach,” SIAM Conference on Geometric Design ’97, Nashville, TN, November 3– 6, 1997, 23.
6. Meyer, Joerg, “Interactive Visualization of Medical and Biological Data Sets,” Ph. D. thesis; Shaker Verlag, Germany, 1999.

7. Reddy, D.; Rubin, S., “Representation of Three-Dimensional Objects,” CMU-CS-78-113, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1978.
8. Schneider, Timna Esther, “Multiresolution-Darstellung von 2D-Schichtdaten in der medizinischen Bildverarbeitung,” thesis; Department of Computer Science, University of Kaiserslautern, Germany, December 1997.