

# Texture-based 3-D Brain Imaging

Sagar Saladi, Pujita Pinnamaneni, Joerg Meyer

Mississippi State University

NSF Engineering Research Center, 2 Research Blvd., Starkville, MS 39759

{saladi | pp2 | jmeyer} @ erc.msstate.edu

## Abstract

Different modalities in biomedical imaging, like CT, MRI and PET scanners, provide detailed cross-sectional views of the human anatomy. The imagery obtained from these scanning devices are typically large-scale data sets whose sizes vary from several hundred megabytes to about one hundred gigabytes, making them impossible to be stored on a regular local hard drive. San Diego Supercomputer Center (SDSC) maintains a *High-Performance Storage System (HPSS)* where these large-scale data sets can be stored. Members of the National Partnership for Advanced Computational Infrastructure (NPACI) have implemented a *Scalable Visualization Toolkit (Vistools)*, which is used to access the data sets stored on *HPSS* and also to develop different applications on top of the toolkit. 2-D cross-sectional images are extracted from the data sets stored on *HPSS* using *Vistools*, and these 2-D cross-sections are then transformed into smaller hierarchical representations using a wavelet transformation. This makes it easier to transmit them over the network and allows for progressive image refinement. The transmitted 2-D cross-sections are then transformed and reconstructed into a 3-D volume. The 3-D reconstruction has been implemented using texture-mapping functions of Java3D. In a typical application, the user might be interested in a certain section of the data set (region of interest). For example, when a physician wants to examine a tumor in the brain, he or she needs to visualize the section of the tumor in the brain at a high resolution. For this we need to extract sub-volumes of the data set. These sub-volumes are then transmitted and rendered at a higher resolution than the rest of the data set.

## 1. Introduction

Medical imaging devices, such as CT scanner, MRI scanner and PET scanner, provide detailed image information of various parts of a human or animal body. Those kinds of scanners are used in veterinary and human biology and medicine. They provide comprehensive 2D cross-sectional images of a selected region of a body, which helps in investigating a pathologic condition or a disease. CT scans are a well-established medical imaging technique for studying stroke, trauma and tumor conditions. The image series obtained from these scanning devices are large-scale data sets whose sizes range from a few

megabytes to several gigabytes (7MB – 100GB). The sizes of these data sets make them difficult to be stored on a local hard drive and to be transmitted over currently existing networks. SDSC maintains a large-scale data repository where these data sets can be stored. Access to these data sets is provided to biologists and other researchers all over the world in closed or open user groups, and the data are transmitted in small, compressed portions upon request by a client application. The transformation of large volumes into compact sub-volume representations enables faster transmission over the network. The details on accessing these data sets and transforming them into a more compact representation using a Haar wavelet transformation are discussed below and in the following sections.

The *Scalable Visualization Toolkits (Vistools)*, an NPACI initiative, primarily developed at SDSC, the Scripps Institute (La Jolla, CA), UC Davis, U Texas, and Mississippi State University, support a variety of different file formats to store all kinds of structured and unstructured meshes. The file format used for biomedical 3-D image data is called the *vol* file format, which supports regular three-dimensional grids. The *vol* format can be encoded into three different formats, namely *vols*, *volb*, and *volc*. If a data set contains 8-bit scalar values, it is written as a *vols* file. If a data set contains RGB or RGBA values, it is written as in *volb* format (32 bits). Each byte represents a color channel. If no alpha channel is present, the alpha value is zero. 64-bit image data is written in *volc* format. A *volc* file stores element data as 64-bit RGB-alpha-beta values. The color component values are truncated to 10 bits for red, 12 bits for green and 10 bits for blue. The alpha and beta components are truncated to 16 bits each. The *vol* file format also supports a chunked layout scheme in which data is arranged in smaller subsets, which allow for sub-volumes to be extracted. In a non-chunked layout, the data is typically arranged in slices (*x/y* planes), which makes it difficult and inefficient to access data from adjacent slices, i.e., along the *z* axis. In a chunked layout, data are stored as sub-volumes, which makes it easier to access voxels in adjacent slices. This is of particular importance when rendering a region of interest at a higher resolution than the rest of the image, which is only provided as low-resolution context information. A designated sub-volume, which comprises the entire data set, is stored at a lower resolution to provide this context information.

On the client side, the wavelet-transformed data are transformed back from local frequency domains into a spatial domain. Once the selected area of the original volume is reconstructed for a given level of detail, it is rendered as a 3-D volume using texture-mapping techniques in Java3D.

## **2. Background**

Volume rendering of biomedical data sets in 3-D is important for physicians and biologists to get a detailed view of the anatomy of a human or animal subject. Rendering algorithms based on raytracing have been implemented [9,13], which turned out to be very time-consuming for large-scale data sets.

More advanced implementations are based on some of the hardware-acceleration features of OpenGL [8], Open Inventor and Java [1,10]. Hardware-based rendering has been implemented to provide interactive real-time rendering capability for volumes under certain restrictions (limited volume size, limited flexibility in terms of rendering algorithms and illumination) [14]. Very few approaches deal with volume rendering for web-based applications. Previous work includes slicing of large volumetric data sets done locally on the client side [4] or remotely on the server side [11]. Several methods have been developed for lossy compression of gray level data [13]. Other authors suggest to download an entire data set, which is not always feasible with limited storage capacity on the client side. Finally, server-based can be inefficient if large images need to be transmitted to the client side, making interactive rendering and higher frame rates (3-10 frames per second) impossible [16]. In order to overcome these difficulties, we have implemented a client-based rendering scheme in Java3d, which is platform-independent and scalable. This means that data sets in the order of 100 GB can be rendered on the client side at reasonable resolution and with interactive frame rates from various perspectives. Data compression and efficient data storage techniques on the server side are essential for this algorithm.

Various image compression schemes like JPEG, Multiwavelet image compression techniques [12], and Multi-resolution Trees [17] have been discussed in the past. In most of these cases, the compression has only been discussed in two dimensions. In our method we are using a 3-D Haar wavelet transformation, which enables interactive volume rendering through efficient decomposition and rapid reconstruction of 3-D image data. Wavelets are local in both frequency and spatial domains. Therefore, they preserve features, such as defined gradients, abrupt discontinuities and sharp spikes, as they occur in high-resolution image data, in a more compact way than in uncompressed format, and still achieve a reasonable approximation [2]. The data is stored as a sum of low-resolution image data and a set of detail coefficients. These detail coefficients usually have very small values and can be quantized or neglected in order to achieve better compression rates. The quantization thresholds determines the amount of detail lost in the image. If the data is not quantized, lossless compression can be achieved, by still providing better compression rates due to the smaller values of the detail coefficients. A wavelet-transformed uncompressed volume occupies the same amount of space than a non-transformed volume. It provides a hierarchy of several levels of detail, if the transformation is applied repeatedly. These features make wavelets an excellent compression and efficient data storage tool. Wavelets have also been proved advantageous in terms of computational complexity and image quality of compressed images [11].

Our approach deals with extracting and transforming cross-sections and sub-volumes on the server side, transmitting the required sub-volume in high resolution and the rest of the data as contextual information, and reconstructing the 3-D volume on the client side using texture mapping [3,4] in Java3D.

We have used Java as our programming language to implement our visualization system since it is platform independent and most suitable for web-based applications.

### 3. Implementation

To make use of enhanced storage capabilities, the data sets are stored on a *High-Performance Storage System (HPSS)*, which allows the user to define closed and open user groups, and multiple copies or versions of a file at different locations. The actual storage pattern is transparent to the user. We are accessing the data from *HPSS* using NPACI's *Scalable Visualization Toolkits*. The data are retrieved from the repository as subsets of 2-D cross-sections, which are then assembled into 3-D sub-volumes. These sub-volumes are then transformed using the Haar wavelet transform and compressed into a more compact representation. These data packets are then transmitted over the network.

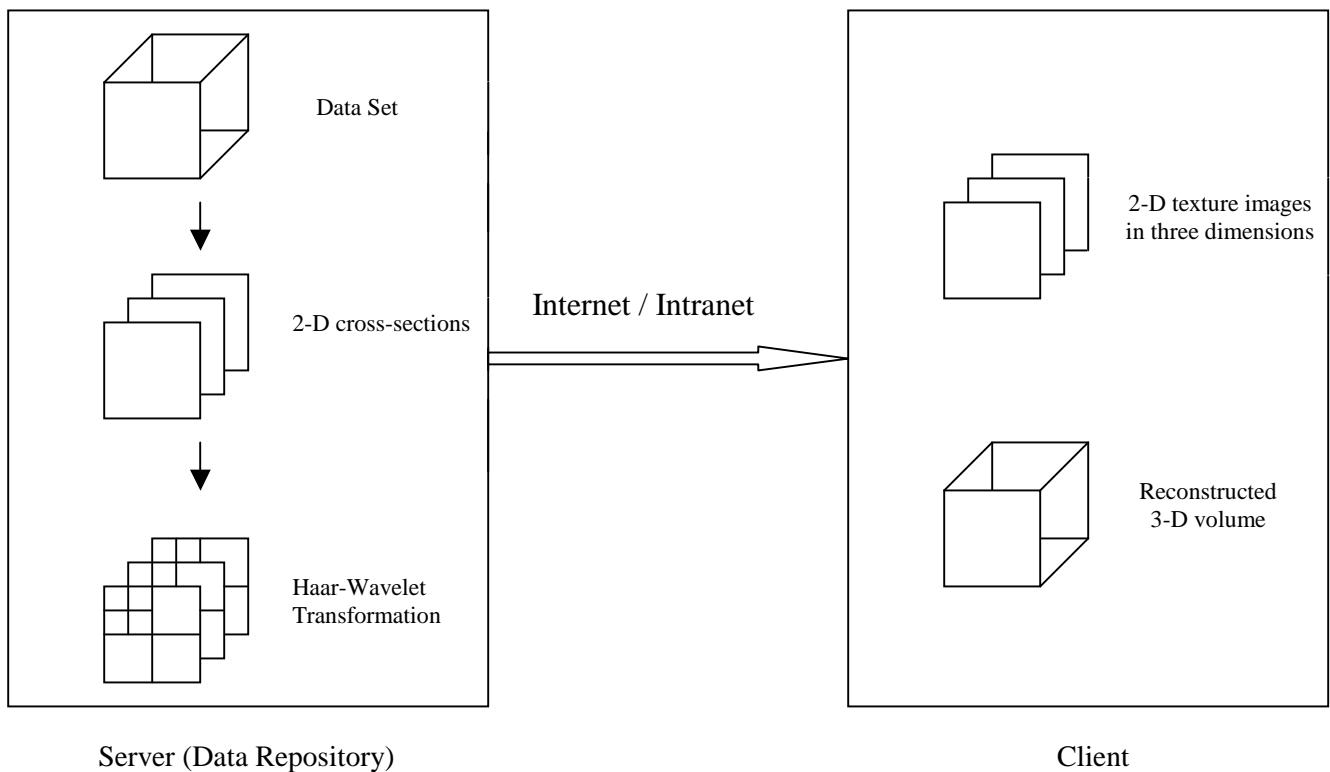


Figure 1: Visualization System

On the client side, the transformed wavelet packets are uncompressed and then reconstructed into a 3-D volume using texture-mapping techniques in Java 3D. Textured planes are created in three perpendicular directions in order to obtain a three-dimensional image, which can be viewed from different directions. The number of planes is determined by the performance of the client application. Currently, approximately 50 planes can be rendered in real-time, providing sufficient image quality and rendering speed.

For a detailed study of a region of interest, for instance a particular section of the brain, sub-volumes can be extracted from the data set. For example, a user might only be interested in a tumor or some other pathological tissue. Then the region can be marked in the Java application using a mouse-driven 3-D cursor, a request is sent to the server, and the server extracts the respective set of sub-volumes from the data repository and transmits to the client. These sub-volumes are typically rendered at a higher resolution than the rest of the data set. Our 3-D wavelet compression schemes allows for different levels of detail and progressive image refinement.

## **4. Data extraction, compression, and rendering**

### **4.1 Extraction of 2-D cross-sections from volume data sets**

NPACI's *Scalable Visualization Toolkits (Vistools)* are available in Java and C++ versions. We have used the Java version to extract a series of 2-D cross-sections from a CT scan of a human brain (*ctbrain.vols*). The data set consists of 512 x 512 x 231 elements. The size of the file was 60,555,264 bytes (approx. 57.8 GB). 2-D cross-sections have been extracted in  $x$ ,  $y$  and  $z$  directions. The *Scalable Visualization Toolkits* provide a method that reads a range of elements from a given data set and returns their values in the host's byte order and word size. This method implements a decoder, which reads the data from a file and decodes its specific format. The decoding is transparent to the user. *VisTools* automatically recognizes the file type by identifying the magic number at the beginning of a file (*vols*, *volb* or *volc*). The resulting byte stream is stored in a buffer.

### **4.2 Extraction of sub-volumes from volume data sets**

It turned out that a slice-based storage layout is not the best way for extracting sub-volumes. In most cases, the user might not be interested in the whole data set, or might not need to see the entire data set at maximum resolution. For example, a surgeon might be interested in viewing a brain tumor and the surrounding tissue in greater detail, while the location and context information can be rendered at a lower

level of detail. To handle such situations, sub-volumes of the data set need to be extracted [6,7]. *Vistools* also provides a method for a chunked storage layout, which can be used to extract sub-volumes from a data set. This method computes and returns a one-dimensional memory index corresponding to the given  $n$ -dimensional grid coordinates in the data set, which is stored in a chunked format. These one-dimensional memory indices are used to read from the file using the same method that was used to extract a cross-section of the data set. We used a file that has been stored in chunked format to extract a series of adjacent sub-volumes. Our sample file is again a CT scan of a human brain (*ctbrain\_c32.vols*).

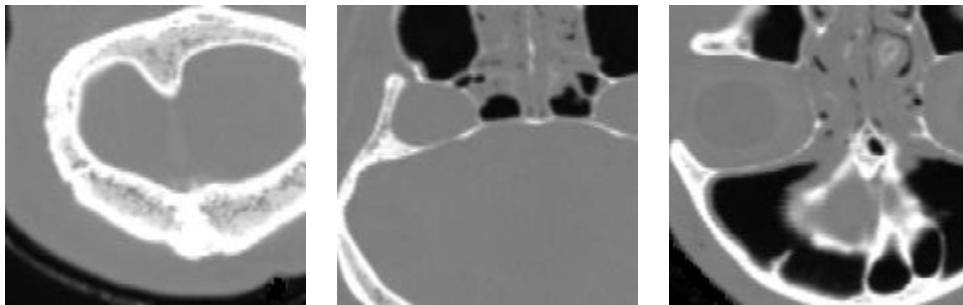


Figure 2: Different cross-sections of a sub-volume

### 4.3 Haar Wavelet Transformation

After the cross-sections or sub-volumes have been extracted, the data are transformed using the Haar wavelet transformation. The Haar wavelet transformation has a local frequency domain and can be computed effectively using integer arithmetic. The procedure can be easily implemented. To provide a better idea about the implementation of a Haar wavelet transformation, we illustrate the algorithm with a simple example.

Assume we have a one-dimensional image with an eight pixel resolution, where the pixels have the following values [5]:

7   5   3   9   3   7   5   3

By applying the Haar wavelet transformation we can represent this image in terms of a low-resolution image and a set of detail coefficients. The transformed data coefficients are obtained by averaging two consecutive pixels, while the detail coefficients represent the difference between the average and one of the two consecutive pixels. So, the above image will be represented as follows after the first cycle:

Low-resolution image: 6   6   5   4      Detail coefficients: 1   -3   -2   1

The original image can be represented as a low-resolution image  $((a+b)/2)$ , which consists of four pixels, and another four-pixel image, which contains the detail coefficients  $((a-b)/2)$ . Recursively iterating this algorithm leads to an image that is reduced by a factor of two for each cycle. The detail coefficients are needed for reconstructing the transformed image.

This simple implementation can be lifted to higher dimensional cases. For a 2-D wavelet transformation, the algorithm is applied first in  $x$ -direction, and then in  $y$ -direction. Similarly, for a 3-D wavelet transformation, the algorithm is applied in  $x$ -,  $y$ - and  $z$ -direction successively. One cycle for an  $n$ -dimensional data set is defined as the completion of the algorithm in all  $n$  directions.

As mentioned earlier, large-scale data sets in the order of several gigabytes are considered for this transformation technique. Therefore, one of the design goals for our Java implementation was scalability. Precautions have been taken not to exceed the amount of main memory available both on the server and the client side. The number of slices loaded into main memory is limited, thus allowing to apply the wavelet transformation successively to adjacent slices. The data structures used to store these data sets are 3-D arrays, where the array sizes are expressed in powers of two. The original resolution of the images is converted into the next larger power of two and the array dimensions adapt accordingly. For the data set we used in the above example, the 3-D array defined is  $512 \times 512 \times 256$ . For the first cycle, the transformation algorithm is first run along the  $x$ -direction, row by row ( $y$ -direction), for each of the 231 slices ( $z$ -direction). While loading the slices into the 3-D array, care is taken to fill in the extra space allotted by zeros.

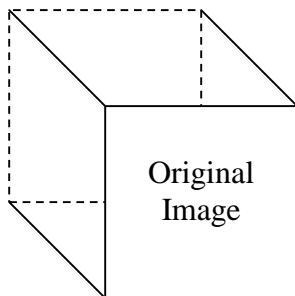


Figure 3: Original image array

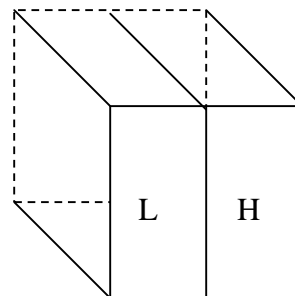


Figure 4: First run:  $x$ -direction

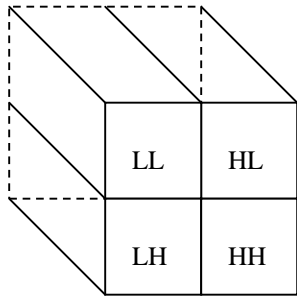


Figure 5: Second run: y-direction

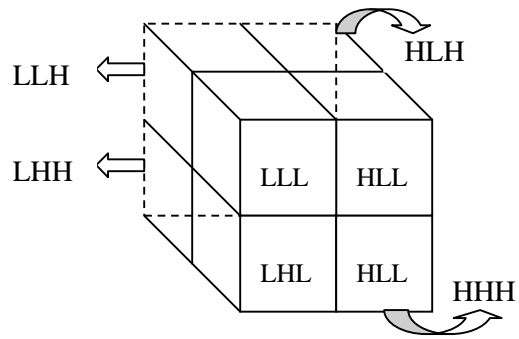


Figure 6: Third run: z-direction

As shown in the figures above, the image array is split into two halves containing the transformed data and the detail coefficients. The transformed data coefficients are the low-pass filtered components, while the detail coefficients represent the high-pass filtered volume components. After transforming the data set in  $x$ -direction (Figure 4), this 3-D array is then transformed along the  $y$ -direction (Figure 5). Finally, the 3-D array is transformed along the  $z$ -direction. The resulting 3-D array is shown in Figure 6.

The final array structure gives us a low-resolution version of the 3-D volume (the LLL segment in one sub-octant, and the detail coefficients, which are used for the reconstruction, in the other seven octants and in the sub-octants of the next higher levels). The indexing of the array is defined such that the transformed coefficients are stored in the upper-left-front corner for each successive cycle. For each cycle, the size of the original volume is reduced by a factor of eight. During consecutive cycles, only the upper left portion of the transformed volume is considered, leaving the detail coefficients intact. Figure 7 shows the result after two cycles.

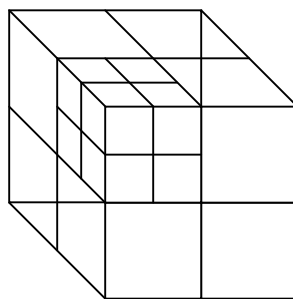


Figure 7: After two cycles



#### 4.4 Reconstruction of the wavelet-transformed data set

During transmission, the low-pass filtered coefficients, which occupy the upper-left-front octant of the 3-D array, are sent first, while the detail coefficients are transmitted at a later time when the network is idle. When the detail coefficients are received on the client side, detail information is added to the volume, which has already been rendered, in order to refine the image. Our texture-based approach allows for gradual refinement of 2-D texture images, so that existing data does not need to be modified. The detail pixels can simply be added to the texture plane images, replacing the replicated pixel values that were stored previously. The reconstruction of the image data uses simple arithmetic operations (integer arithmetics). Therefore, the implementation is very efficient and can be executed in real time by the Java applet. As the array received on the client side consists of the low-pass and detail coefficients, the respective pixel values at each cycle are obtained by adding and subtracting the corresponding detail coefficients to and from the low-pass filtered coefficients. To illustrate this technique, let us go back to the one-dimensional image, which we considered for explaining the transformation process. Now we have the transformed and detail coefficients as given below:

Low-resolution image: 6   6   5   4                      Detail coefficients: 1   -3   -2   1

To obtain the pixel values of the next higher level of detail, we add the low-pass and detail coefficients to obtain one pixel of the first pixel pair, and subtract them to get the second pixel of each pair.

The original pixel values are: 7   5   3   9   3   7   5   3

During reconstruction, it is sufficient to render a user-selected sub-volumes at the highest possible resolution, while the rest of the data is reconstructed at a lower resolution to provide contextual information about the location of the region of interest.

#### 4.5 Reconstruction using Texture mapping

Texture mapping is a technique to add a visual structure to a geometric object. We create a polygon and map a texture image onto it by specifying the texture image location and setting the texturing attributes in Java3D. The cross-sections that have been reconstructed in  $x$ ,  $y$  and  $z$  direction on the client side are used to create a set of three series of perpendicular planes in 3-D space. These cross-sections are not the same

as the original slices because the data set can be sliced in arbitrary direction. An interpolation technique is used to create additional slices, if necessary. A *TextureLoader* utility class in Java3D is used to load the texture images. These images are then mapped in back-to-front order onto a series of polygons arranged in parallel in  $x$ ,  $y$  and  $z$  directions as shown in figure 8. 2-D texture mapping in Java3D has been used to achieve this. All the polygons are drawn as parallel planes, and the 3-D texture coordinates are chosen accordingly.

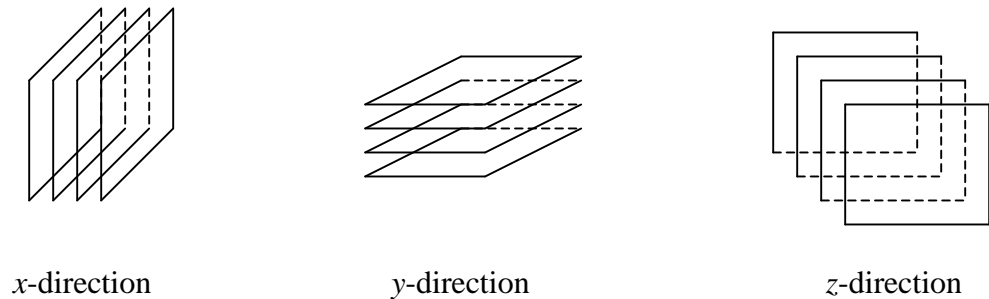


Figure 8: Texture images mapped onto a stack of polygons

Consider the case of mapping 2-D cross-sections in  $z$ -direction. Since the cross-sections are mapped one in front of the other, perpendicular to the viewer, only the first cross-section would be visible while the other sections are occluded. To make the remaining cross-sections visible, transparency values are assigned to all rectangular planes that are used for texture mapping. The resulting semi-transparent images of the data set reveal the interior structures of an object. Java3D supports two modes of transparency: a scalar transparency value that can be applied to the geometric object, i.e., the textured plane, as a whole, or a binary transparency value that can be assigned to each pixel of the texture image. The following results were obtained by using the per-plane transparency mode (Figures 9 and 10).



Figure 9: A 3-D view of the ear



Figure 10: A 3-D view of a partial data set

Instead of assigning the same transparency value to an entire texture plane, we can use binary transparency to eliminate the background pixels and keep the rest of the slice intact. Removing all the background pixels that fall below a certain threshold from the 2-D cross-sections and using them for the texture maps creates a 3-D volume. To access and modify the alpha channel of the RGBA pixel values of the 2-D cross-sections, Java2D *BufferedImage* and *ColorModel* classes have been used. Different values are assigned to the alpha component of the pixels constituting the skull and to the background pixels. The background pixels are distinguished from the rest of the image by using a certain threshold. In order to enable binary transparency for dark or extremely bright (white) background pixels (CT/MRI vs. cryo-sections), a range can be defined for the background pixels. During the rendering process, the alpha component of each pixel is checked. If the value is greater than the threshold value, then the pixel is drawn, otherwise the pixel is not drawn. By using varying the threshold, a simple segmentation algorithm can be implemented which allows us to select different materials, such as skin or bone. To implement this mode, we used the *RenderingAttributes* class.

## 5. Results

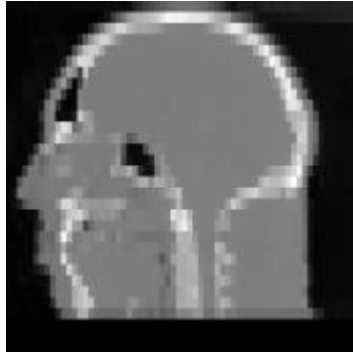
We applied our Haar wavelet transform algorithm to several test data sets. Figure 11 shows a typical example of a CT scan of a human brain (512 x 512 x 231). The image series shows a cross-section of the 3-D volume at four different levels of detail.



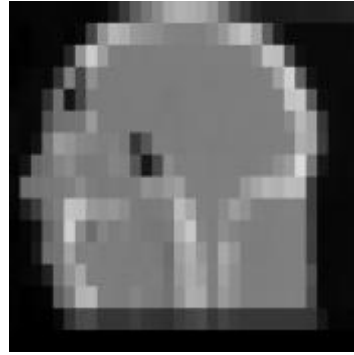
1<sup>st</sup> iteration



2<sup>nd</sup> iteration



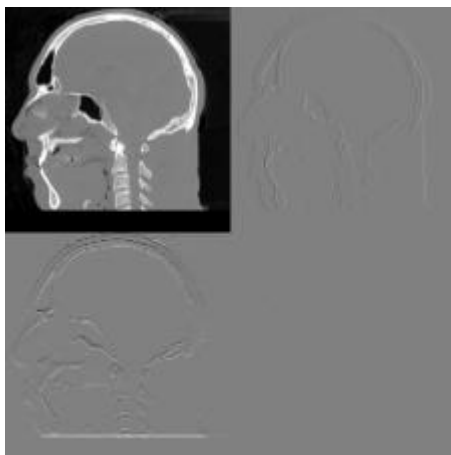
3<sup>rd</sup> iteration



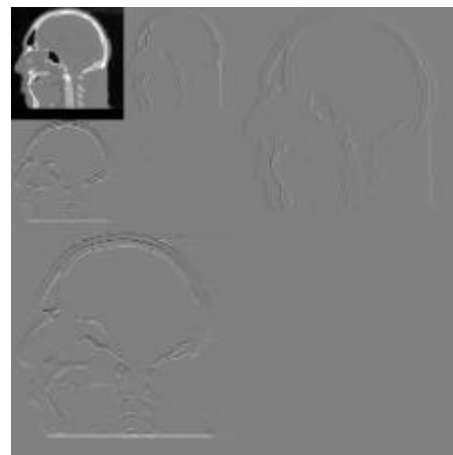
4<sup>th</sup> iteration

Figure 11: Compressed images in successive iterations

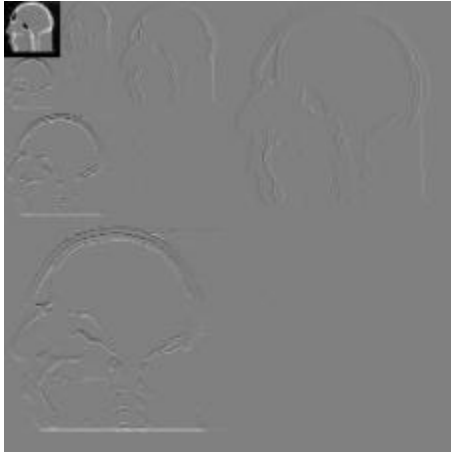
Successive iteration reduces the size of the original volume by a factor of eight ( $2^3$ ). By applying a run-length encoding scheme, we were able to obtain much better compression rates for our 3-D algorithm than for a 2-D algorithm. This scheme made it possible to transmit a low-pass filtered data set from a server to a rendering client over a medium-bandwidth network and render it in real time [15]. Detail coefficients are transmitted successively to improve the resolution and quality of the 3-D reconstruction by adding more information to the texture planes. The data set is stored on a server with large storage capacity, while the Java3D based rendering client runs on a desktop machine.



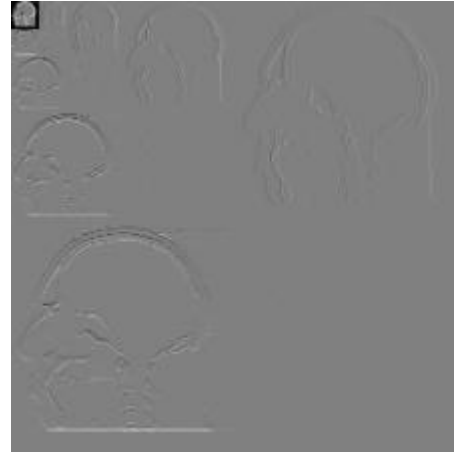
1<sup>st</sup> iteration



2<sup>nd</sup> iteration



3<sup>rd</sup> iteration



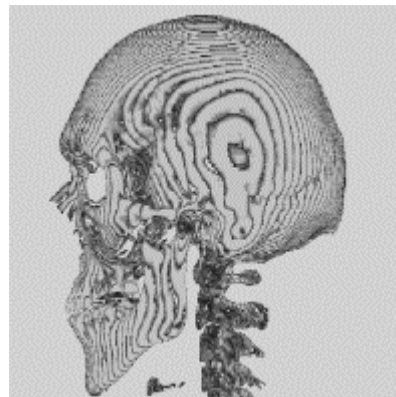
4<sup>th</sup> iteration

Figure 12: Compressed image data and detail coefficients after four iterations

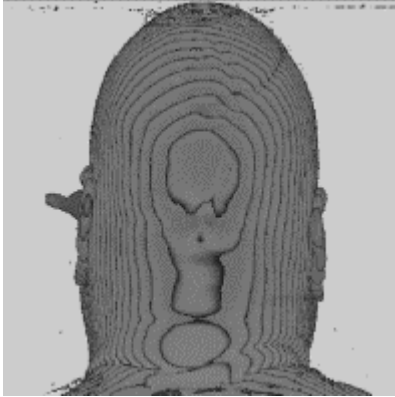
By using 2-D texture mapping in Java3D in binary transparency mode for the image planes, a 3-D volume has been obtained. Results are shown below (Figure 13).



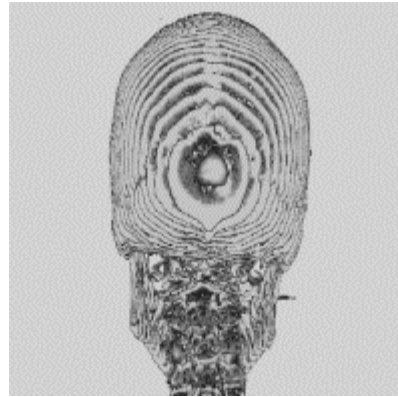
Sagittal view of the volume with skin



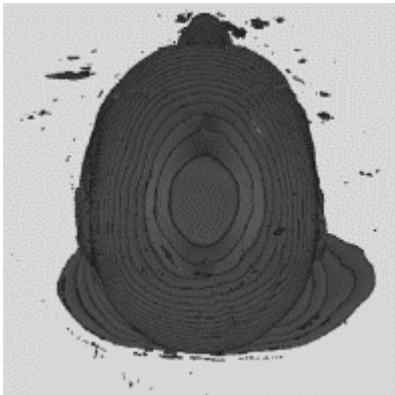
Sagittal view of the volume without skin



Coronal view of the volume with skin



Coronal view of the volume without skin



Transverse view of the volume with skin



Transverse view of the volume without skin

Figure 13: Different views of a 3-D reconstructed volume

## 6. Conclusions

The total CPU time taken to read the entire data set from the data repository has been analyzed [15], and an optimal cache size of 32 bytes has been determined, which is a hardware-dependent variable for the *Scalable Visualization Toolkit*. The Java-based system is hardware-independent and has been tested on SUN Ultra Sparc and SGI O2 (R12000, RISC architecture, 256 MB RAM) workstations. The image quality of the transformed volume has also been satisfactory (Figures 9, 10, and 13).

We have presented a platform-independent scalable wavelet transformation and data compression algorithm and a hierarchical, progressive, texture-based Java3D data reconstruction and rendering method. Future work includes enhancing the visibility of interior structures occluded by the surface of the object. This can be achieved by using adaptive per-pixel transparency, which is currently not supported in Java3D. Therefore, the application will be ported to C/C++, making use of the C++ version of the *Scalable Visualization Toolkits*, which will also allow for higher data access and transfer rates, and for faster image refinement during visualization.

## Acknowledgements

We thank Arthur J. Olson (The Scripps Institute, La Jolla, CA) for providing the sample data sets. We also thank David Nadeau and Jon Genetti (San Diego Supercomputer Center, SDSC) for providing additional information and the source codes of the *Scalable Visualization Toolkits*. This project was funded in part by the National Partnership for Advanced Computational Infrastructure (NPACI) under award no. 10195430 00120410.

## References

- [1] M. Bailey, "Interacting with Direct Volume Rendering," IEEE Computer Graphics and Applications, Vol. 21, Issue 1, pp. 10-12, February 2001.
- [2] Brani Vidakovic and Peter Müller, "Wavelets for kids," A Tutorial Introduction, Institute of Statistics and Decision Science, Duke University, Durham, NC, 1991.
- [3] B. Cabral, N. Cam and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," ACM Symposium on Volume Visualization, pp. 91-98, 1994.
- [4] K. Engel, P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl, "Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications," IEEE Visualization 2000, Proceedings, pp. 449-452, 587, October 2000.
- [5] Eric J. Stollnitz, Tony D. DeRose and David H. Salesin, "Wavelets for Computer Graphics: A Primer Part 1," IEEE Computer Graphics and Applications, May 1995.
- [6] P. Hastreiter, B. Tomandl, K. Eberhardt, and T. Ertl, "Interactive and Intuitive Visualization of Small and Complex Vascular Structures in MR and CT," Engineering in Medicine and Biology Society, Proceedings of the 2<sup>nd</sup> Annual International Conference of the IEEE, Vol. 2, pp. 532-535, November 1998.

- [7] A.K.W. Law, H. Zhu, F.K. Lam, F.H.Y Chan, B.C.B Chan and P.P. IU, "Tumor boundary extraction in multislice MR brain images using region and contour deformation," Proceedings of International Workshop on Medical Imaging and Augmented Reality, pp. 183-187, June 2001.
- [8] Z. Lee, P. Diaz and E. Bellon, "Manipulation of volume and graphics objects for the visualization of medical data under OpenGL," Proceedings of the Eighth IEEE Symposium on Computer-Based Medical Systems, pp. 89-93, June 9-10, 1995.
- [9] Marc Levoy, "Efficient Ray Tracing of Volume Data," ACM Transactions on Graphics, Vol. 9, No. 3, pp. 245-261, July 1990.
- [10] M. Meissner, U. Hoffmann and W. Strasser, "Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering Using OpenGL and Extensions," IEEE Visualization '99, Proceedings, pp. 207-526, October 1999.
- [11] Meyer, Joerg, Ragnar Borg, Bernd Hamann, Kenneth I. Joy, and Arthur J. Olson, "VR based Rendering Techniques for Large-scale Biomedical Data Sets," Online Proceedings of NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization, Granlibakken Conference Center, Tahoe City, CA, pp. 73-76, October 15 - 17, 2000.
- [12] Michael B. Martin and Amy E. Bell, "New Image Compression Techniques Using Multi-wavelets and Multiwavelets Packets," IEEE Trans. Image Processing, Vol. 10, pp. 500-510, April 2001.
- [13] T. Miyazawa, R. Yoshida, M. Kimura and T. Otsuki, "Visualization of 3D medical images for radiotherapy planning," Nuclear Science Symposium and Medical Imaging Conference Record of the 1991 IEEE, Vol. 3, pp. 1553-1557, November 2-9, 1991.
- [14] Hanspeter Pfister, "Cube-4: A Scalable Architecture for Real-Time Volume Rendering," ACM/IEEE Symposium on Volume Visualization, pp. 47-54, 1996.
- [15] Sagar Saladi, "Texture-based 3-D Reconstruction of Biomedical Data Sets," Tri-State Engineering Society Meeting, Destin, FL, June 2001.
- [16] C.T. Silva, A.E. Kaufman and C. Pavlakos, "PVR: high-performance volume rendering," IEEE Computational Science and Engineering, vol. 3, pp. 18-28, Winter 1996.
- [17] Wilhelms, Jane, and Allen Van Gelder, "Multi-dimensional Trees for Controlled Volume Rendering and Compression," 1994 ACM Symposium on Volume Visualization, pp. 27-34, October 1994.
- [18] K.J. Zuiderveld, P.M.A. van Ooijen, J.W.C. Chin-A-Woeng, P.C. Bujis and F.H. Ol Posti, "Clinical evaluation of interactive volume visualization," Proceedings of Visualization, pp. 367-370, October 27 – November 1, 1996.