Segmentation and Texture-based Hierarchical Rendering Techniques for Large-scale Real-color Biomedical Image Data

Joerg Meyer
Mississippi State University, Department of Computer Science
310 Butler Hall, Mississippi State, MS 39762-9637
jmeyer@cs.msstate.edu

Ragnar Borg, Ikuko Takanashi, Eric B. Lum, Bernd Hamann
University of California, Davis, Department of Computer Science
CIPIC – Center for Image Processing and Integrated Computing
One Shields Avenue, Davis, CA 95616-5270
Ragnar.Borg@proxycom.no, ikuko@ucdavis.edu, lume@ucdavis.edu, bhamann@ucdavis.edu

Key words: Biomedical Imaging, Volume Rendering, Interactive Rendering, Virtual Reality

Abstract: Hierarchical, texture-based rendering is a key technology for exploring large-scale data sets. We describe a framework for an interactive rendering system based on a client/server model. The system supports various output media from immersive 3-D environments to desktop based rendering systems. It uses web-based transport mechanisms to transfer the data between the server and the client application. This allows us to access and explore large-scale data sets from remote locations over the Internet. Hierarchical space-subdivision, wavelet compression, and progressive data transmission are used to visualize the data on the client side.

1.   INTRODUCTION

We present a framework for distributed hierarchical rendering of large-scale data sets that addresses two problems at the same time: (i) limited network bandwidth and (ii) limited rendering resources. Our goal is to convert the

data set into a more compact representation and to break it down into smaller subunits, while making effective use of multiresolution techniques. Our system uses a server-based data repository, which can be accessed remotely by a rendering client in a hierarchical way. This is beneficial both for effective data transmission over the Internet and for optimizing rendering time on the client side.

The system consists of two components: (i) a server, which serves as a data repository for large-scale data sets, and (ii) a rendering client, which accesses the data on the server and renders the data (figure 1). For efficiency reasons, most of the rendering is done on the client side. This means that the client uses specially designed Java applets provided by the server to render the data. Depending on the type of request that is sent by the client, the server returns a Java applet for rendering geometry data (iso-surfaces) or volume data. A 2-D texture-based desktop rendering application has been implemented in Java3D. An alternative rendering client that uses 3-D texture-based rendering and OpenGL has been implemented in C++ [Ope99]. This client can run in desktop mode or in VR stereo mode.
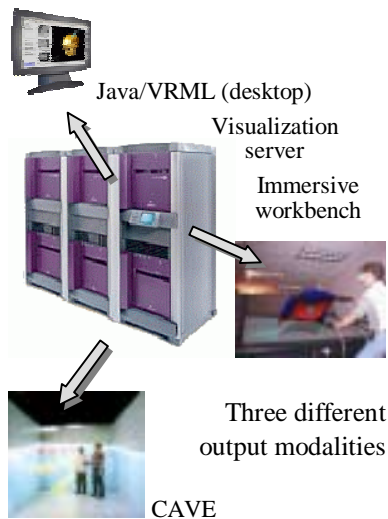
Customizing the rendering application keeps the applet small and avoids additional overhead for different cases. The initial data set is also small. The first scene provides the user with a preview of the entire data set and is refined later upon subsequent requests by the client. The term 'scene' in this context refers to a 3-D object, which can be rotated, scaled, and translated interactively, as opposed to a single 2-D preview image. All data transmitted from the server is rendered as a 3-D object.

In order to keep the amount of data transmitted over the Internet small, the algorithm uses hierarchical data compression and storage techniques. This includes a data preparation step (section 3), adaptive space subdivision (delimited octrees for volume), wavelet-based data reduction and storage of large volume data sets, and progressive data transmission techniques for hierarchically stored volume data sets. These hierarchical data structures will be explained in section 5.

Java/VRML (desktop)

Visualization server

Immersive workbench

Three different output modalities

CAVE

Figure 1: Client/server system

The web-based user interface combines HTMLform-driven server requests with customized Java applets, which are transmitted by the server to accomplish a particular rendering task (section 5). This allows us to create a

flexible user interface, which can be easily modified and customized. It also enables features such as hypertext-documentation and online-help. The C++ version features interactive rendering and improved rendering performance.

2.   RELATED WORK

In [Mic97], the authors present a framework for an interactive rendering client in Java. In contrast to this universal solution, we use customized Java applets that are loaded on demand (volume rendering, geometry rendering, hybrid rendering, other volume tools). Similar to [Tra97], the tools are available on the Internet as a rendering service for large-scale data sets.

Other web-based approaches include rendering of data directly from the compression domain (volume data) [Lip97] and isosurface extraction techniques (geometry data) [Eng99]. We present a method that combines these techniques with an efficient hierarchical data structure. Our method takes advantage of the fact that reconstruction from the Haar wavelet domain can be implemented very efficiently using integer arithmetic without additional overhead on the client side. The resulting patterns can be written directly into the texture buffer.

Our prototype implementation features 2-D/3-D preview capability (both in the Java3D and in the VR version), interactive cutting planes in a 3-D rendering, hierarchical iso-surface models to provide context information [Lor87, Nie91, Mon94], a lens function to examine particular regions of interest (ROIs), and variable magnification and lens shape with interactively modifiable ROI. The user interface can be extended to provide additional functionality. Complex geometry scenes are converted into VRML2 so that the web-based client can render the scene interactively in real time.

3.   SYSTEM OVERVIEW

The server (figure 2) integrates a series data preparation steps. First, each data set is prepared for hierarchical storage using NPACI's Scalable Visualization Toolkit before it is added to the data repository (section 7) [Mey99].

The filter step is described in section 4, followed by the space subdivision and wavelet compression scheme. The data is finally stored in a repository (section 5).

The server sends the data either through a web channel together with a Java applet to the Java/VRML client, or to the C++ 3-D rendering engine (figure 3).

4.   DATA PREPROCESSING AND FILTERING

As datasets become larger and more detailed, they can no longer be segmented exclusively by interactive methods or by hand. Also, they do not compress very well because of lots of redundant data in the unsegmented part of the image. Image parts that would not contribute to the rendering do not need to be stored. Therefore, using segmentation in conjunction with data compression, we obtain much better compression rates and improved data transmission times.

Real-color RGB images, such as cryosections, contain much more information than CT or MRI data: the reso-

Figure 2: Server operations

lution is usually higher (150-500 dpi as opposed to 25 dpi, today's standard, leading to an increase of the data volume by a factor of $20^3 = 8,000$), and the color information provides some additional cues. Therefore, we use an automated segmentation pipeline, which allows us to apply different filters and image processing algorithms to separate the tissue from the surrounding material.
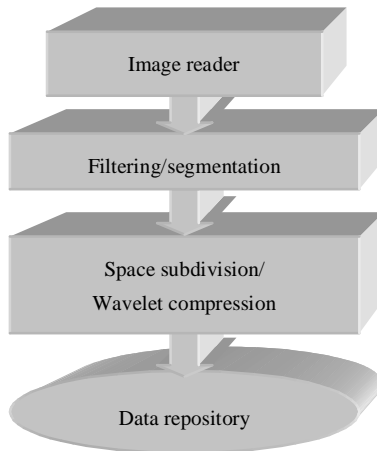
4.1   Human Brain Data Set

The Human Brain Data Set (courtesy of Arthur W. Toga, Ahmanson-Lovelace Brain Mapping Center, UCLA School of Medicine) that we present here has a resolution of 1472 x 1152 pixels per slice and consists of 753 images of cryosections. The data set provides real-color RGB information, 16 bits per channel. Due to the nature of the cutting technique, the data is different from conventional CT or MRI data. The images are not real cross-sections, but merely photographs of the surface and the structures that have not been sliced yet. Cavities or gaps in the brain reveal structures that are actually located behind the cutting plane, i.e., in deeper layers. These parts of the image need to be eliminated and replaced by transparent regions. Also,

the brain must be separated from the surrounding matter, i.e., from the ice, and from the background (figure 4).

4.2   Segmentation: Color Cues

The brain can be easily distinguished from the ice and the background by using the RGB color information from the images. By using a different color model, we can make use of the fact that the red and light brown components are much more dominant in the brain than in the rest of the picture. We have found that the YIQ color model works best for selecting this brown color (the I component detects both the red component and the intensity). This model is used in television for natural representation of skin colors.
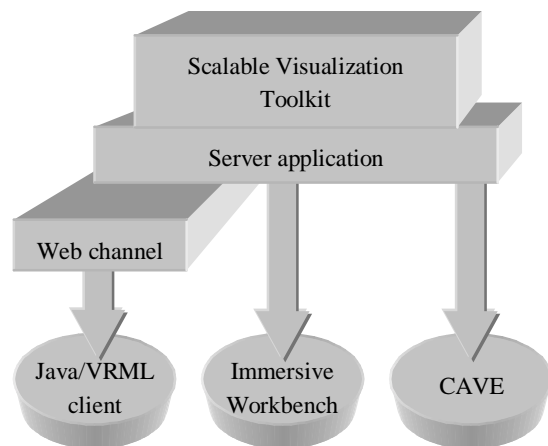
Figure 5(a) shows a resulting image (slice #100 of 753). The brain has been nicely segmented. However, the image still shows transparent spots and holes inside the brain structure and some noise around the brain. To fill these holes, to smooth the contour, and to remove the noise, we apply a set of filters after YIQ thresholding.

Figure 3: Client: access mechanism

4.3   Filtering

Figures 5(b-e) show the individual steps of the filter pipeline. In figure 6, the rectangles represent the filters that were used to fill the holes and smooth the contour, and the ovals depict the filters that were used to remove the noise. Each process shows some progress, but after using a dilation filter, some of the ice appears again close to the contours. An RGB color threshold was used to remove these artifacts.

The segmentation pipeline consists of these steps:

(1) Median filter (5 x 5 window, remove holes, smooth contours);

(2) Region growing (8 directions, check for background pixels, remove noise if below size threshold);

(3) 2D morphology (9 x 9 window, dilation);

(4) RGB thresholding (removal of iced area); and

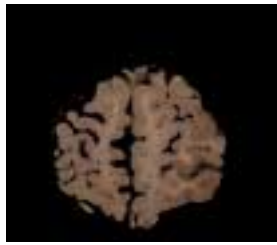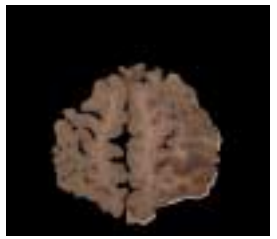(5) 3D morphology (9 x 9 x 9 window, erosion).

Figure 4: Cryosection of   Figure 5(a) YIQ thresholding   (b) Median filter
a human brain
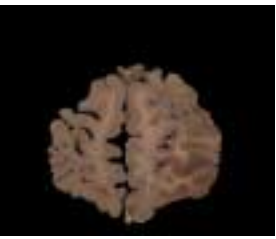
(c) Region growing   (d) 2-D morphology (dilation)   (e) RGB thresholding

Some algorithms require to vary the threshold, because the structure of the image changes across the data set. However, it is much easier to select a set of thresholds than to segment the entire data set by hand.

5.   HIERARCHICAL SPACE SUBDIVISION

Volume data sets are usually arranged by the scanner software as a set of 2-D images, which represent a series of 2-D cross-sections. Putting all those slices together, we obtain a 3-D volume. Unfortunately, when we access the data we typically don't need the implicit coherency across single slices. This coherency stretches only across one preferred direction. Instead, we merely need brick-like coherency within subvolumes. We use a data structure, which uses a combination of delimited octree space subdivision and wavelet compression techniques to achieve better performance.
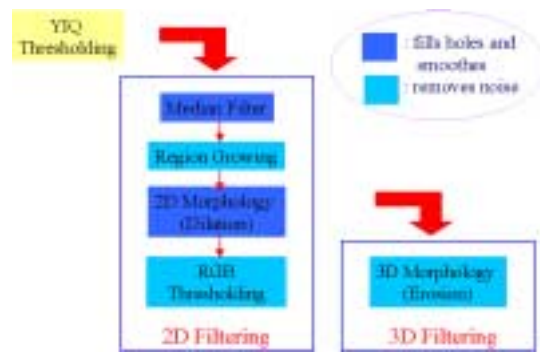
Figure 6: 2-D/3-D filtering

The purpose of this approach is not to present a new compression scheme or a new hierarchical subdivision method. Instead, we present an integrated framework that combines an efficient indexing scheme, a suitable 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 file system to store our derived data structures, and we use filenames as keys to the database, thus avoiding additional overhead, which is typically caused by incorporating additional 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 file system provides the storage system (repository) for the server application, which reads the data at a low resolution from the repository and sends it to a remote rendering client upon request. 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 resolution. This updating procedure typically takes considerably less time, because only a small number of files need to be touched. The initial step, which requires reading the initial section of every file, i.e., of all bricks, can be sped up by storing an additional file which contains a reduced version of the entire data set.

Our new data structure uses considerably less memory than the original data set, even if the user chooses lossless compression (see statistics, section 6). By choosing appropriate thresholds for wavelet compression, the user can switch between lossless compression and extremely high compression rates. Computing time mainly depends on the size of the bricks. Therefore we take a closer look at the filesize to determine an optimal size for the subvolumes.

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. 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. The seek time for a file in general is much higher than the seek time within a file for the desired level of detail.

5.1   File storage

The filesize f for storing the leaves of the octree structure, which is described in section 5.2, should be a multiple n of the minimum page size p of the file system. p is typically defined as a system constant in a system file named `param.h`). n depends on the wavelet compression. 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:

$$f = n \cdot p \geq 8^r \cdot b$$

Both r and b are user-defined constants. Typical values are $b = 512$, which corresponds to an 8 x 8 x 8 subvolume, and $r = 3$, which gives us four levels of detail over a range between 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

$$n \cdot p = 8^r \cdot b,$$

thus
$$n = 8^r \cdot \frac{b}{p}$$

5.2   Delimited octree and wavelet structure

The enormous size of the data sets (section 4) requires to subdivide the data into smaller subunits, which can be loaded into core memory within a reasonable amount of time [Mey97, Mey99]. Since we are extracting subvolumes, it seems quite natural to break the data up into smaller bricks. This can be done recursively by using an octree approach [Jac80, Mea80, Red78]. 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. The memory reduction occurs by skipping the empty regions. Typically, the size of the data set shrinks to about 20%, i.e., one fifth of the original size (section 6).

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. 3-D Haar wavelets fulfill these requirements (figure 7). They also provide the advantage that they can be easily implemented as integer arithmetic. The lower resolution is stored at the beginning of the file, thus avoiding long seek times within the file.
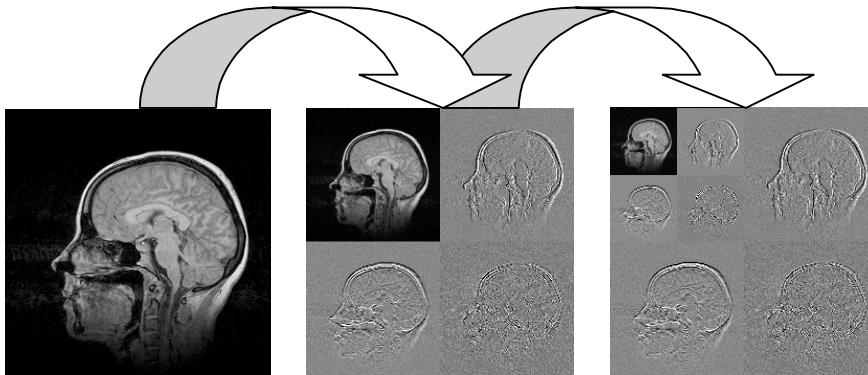

Figure 7: Wavelet compression scheme (2-D case)

Another very useful property is the fact that a volume converted into the frequency domain, i.e., the wavelet representation, requires the exact same amount of memory as the original representation. This is also true for all subsequent wavelet recursions. The wavelet recursion terminates when we have reached a predefined minimum subvolume size b. The lower bound is the size of a single voxel.

Each octant can be described by a number [Fol96, Hun79]. We use the following numbering scheme (figure 8): A leaf is uniquely characterized by the octree recursion 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 that indicates that the file is empty. This prevents us from opening and attempting to read the file and speeds up the computation. The total number of bits is 32.
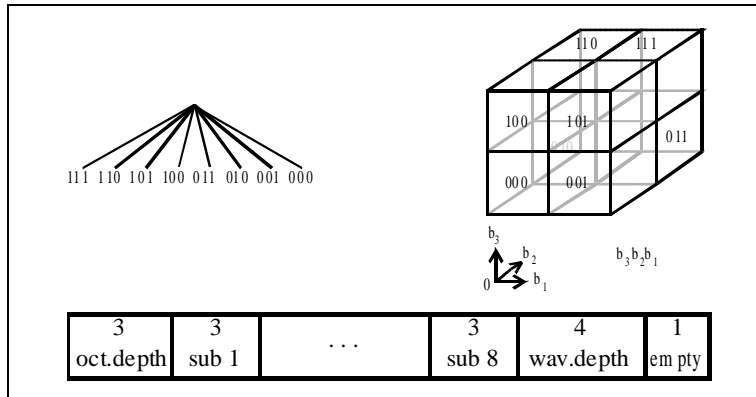
Figure 8: Encoding scheme for file storage

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 concatenating 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 it is 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 recursion 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.

6.   STATISTICS AND RESULTS

Our test applications include molecular biology, medicine, and earthquake simulation. Our prototype for the biomedical field was designed to support three-dimensional visualization of a human brain, which allows us to study details by moving different 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 vastly independent from the size of the data set and requests only as much data as can be displayed and handled by the Java applet. An example of a volume display of a human brain, which can be rendered on the workbench, is shown in figure 9. This image also shows the prototype for a Java applet.

Table 1 shows the reduction of memory which is 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) of the decomposition. The transmission of the full data set ($1024^3 \cdot$ RGB = 3 GB) over a 10Mbps LAN connection took between 232 and 1563 seconds. This data is not representative, because transmission times vary over a great range depending on the network link and the actual network load. The transmission time for the lowest level of detail (64^3) in compressed format was negligible.

Algorithm	Octree			
	level 1		level 2	
Data type	MRI	CT	MRI	CT
Pre-processing	56	63	98	97
Depth	4	4	5	5
Memory				

Table 1: Statistics

The segmented images are volume rendered in hardware using three sets of textures, each set aligned to one of the principle axes. With a Pentium III 550 MHz and a 32M Nvidia GeForce^TM video card, a downsampled 128^3 volume fits in texture memory and can be rendered at over 15 fps. A 256^3 volume does not fit in texture memory and can be rendered at approximately 2 fps. The user can interactively rotate the volume, and apply multiple axis aligned cutting planes to quickly visualize specific regions of the brain (figure 9).
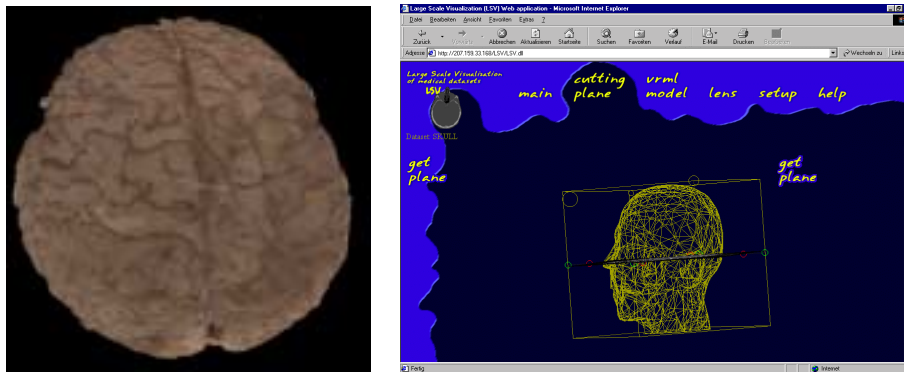
Figure 9: Volume rendering of a human brain (C++ VR client and Java client applet)

The overall conclusion is that the reduction of memory allows for faster response times, more rapid data transmission, and faster rendering. The wavelet decomposition algorithm takes less time than the performance gain by reducing the amount of data to be sent and using an efficient data indexing and storage scheme.

7.   CONCLUSIONS

We have presented a framework for a distributed hierarchical rendering system for large-scale volume data sets. The first step is to filter the data and extract certain features (segmentation). The presented pipeline allows us to segment a large-scale data set semi-automatically with minimal user intervention. The parameters must be chosen carefully in order to obtain optimal results. After filtering and segmenting, the data is converted into a hierarchical format and stored in a repository, which can be accessed by a remote or immersive client application. We have presented an efficient numbering scheme and access method for hierarchical storage of sub-volumes on a regular file system. This method allows us to access a region-of-interest as a set of bricks at various resolutions. The simplicity of the method makes it easy to implement. The algorithm is scalable by increasing the word length for the bit string and the filename length. The algorithm has also been tested on larger data sets. The data decomposition, reconstruction and rendering times for the region-of-interest and the context were still acceptable due to the flat directory structure of the indexing scheme.

Future research focuses on automated image analysis to determine segmentation parameters automatically, the introduction of alternative wavelet compression schemes, support for time-variant data sets, and the use

of a web channel also for the immersive workbench and the CAVE. It would also be desirable to provide an interface to upload data sets and apply filters and data compression algorithms online in an automated way. This would allow users to upload and explore their own data interactively.

The developed tools will be integrated, adapted and evaluated in the National Partnership for Advanced Computational Infrastructure (NPACI) framework. Integration of San Diego Supercomputer Center's High-performance Storage System (HPSS) as a data repository to retrieve large-scale data sets, accessing the data via NPACI's Scalable Visualization Toolkits (also known as VisTools), and evaluation of particular applets with NPACI partners, are some of the main goals.

ACKNOWLEDGEMENTS

REFERENCES

[Eng99] Engel, K.; Westermann, R.; Ertl, T., "Isosurface Extraction Techniques for Web-based Volume Visualization," Proceedings IEEE Visualization '99, 1999, 139–146.

[Fol96] Foley, J. D.; van Dam, A.; Feiner, S. D.; Hughs, J. F.; Philips, Richard L., Computer Graphics: Principles and Practice, Second Edition, Addison-Wesley, 1996.

[Hun79] Hunter, G. M.; Steiglitz, K., "Operations on Images Using Quad Trees," *IEEE Trans. Pattern Anal. Mach. Intell.*, 1(2), April 1979, 145–154.

[Jac80] Jackins, C.; Tanimoto, S. L., "Oct-Trees and Their Use in Representing Three-Dimensional Objects," *CGIP*, 14(3), November 1980, 249–270.

[Lip97] Lippert, L; Gross, M. H.; Kurmann, C., "Compression Domain Volume Rendering for Distributed Environments," Proceedings Eurographics '97, 1997, C95–C107.

[Lor87] Lorensen, William E.; Cline, Harvey E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," ACM Proceedings, Computer Graphics, 21(4), 1987, 163–169.

[Mea80] 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.

[Mey97] Meyer, Jörg; 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.

[Mey99] Meyer, Jörg; Interactive Visualization of Medical and Biological Data Sets; Medizinische Informatik und Bioinformatik, Shaker Verlag, Aachen, 2000, 49–54.

[Mic97] Michaels, Cherilyn; Bailey, Michael, "VizWiz: A Java Applet for Interactive 3D Scientific Visualization on the Web," Proceedings IEEE Visualization '97, 1997, 261–267.

[Mon94] Montani, Claudio; Scateni, Riccardo; Scopigno, Roberto, "A modified look-up table for implicit disambiguation of Marching Cubes," The Visual Computer, 10, 1994, 353–355.

[Nie91] Nielson, G. M.; Hamann, B., "The asymptotic decider: resolving the ambiguity in marching cubes," in: Nielson, G. M.; Rosenblum, L. J. (eds.), IEEE Visualization '91, IEEE Computer Society Press, Los Alamitos, California, 1991, 83–91.

[Red78] Reddy, D.; Rubin, S., "Representation of Three-Dimensional Objects," *CMU-CS-78-113*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1978.

[Ope99] Opengl Architecture Review Board, "The OpenGL Reference Manual – The Official Reference Document to OpenGL, Version 1.2," by Dave Shreiner (ed.), Addison-Wesley, 1999.

[Tra97] Trap, Jens; Pagendarm, Hans-Georg, "A Prototype for a WWW-based Visualization Service," Proceedings Eurographics '97, 1997, 23–30.