Space-Efficient Boundary Representation of Volumetric Objects

Lukas Mroz and Helwig Hauser

VRVis Research Center, Vienna, Austria
`http://www.vrvis.at/vis/`
`{mroz|hauser}@vrvis.at`

Abstract. In this paper we present a compression technique for efficiently representing boundary objects from volumetric data-sets. Exploiting spatial coherency within object contours, we are able to reduce the size of the volumetric boundary down to the size of just a few images. Allowing for direct volume rendering of the down-scaled data in addition to compression ratios up to 250:1, interactive volume visualization becomes possible, even over the Internet and on low-end hardware.

1 Introduction

One major challenge of visualization in general is to deal with a whole lot of data. Especially in volume visualization, common data-sets range between several hundreds of Kilobytes, at the minimum, up to Gigabytes of uncompressed size. In medical visualization, for example, volumetric data-sets of size $256^3 \times 16$ Bit, i.e., 32 MBytes in total, are quite usual. If standard compression like `gzip` [7], for example, is applied, data-sets usually shrink to about 30–60 percent of the original size – still MBytes.

Processing huge data-sets itself poses high-performance requirements on the visualization software, but also storage and transmission of volumetric data-sets easily get into bandwidth problems, especially if multiple data-sets are to be treated. From medical applications, for example, we know that archiving 3D data-sets, which accompany diagnosis data, significantly stresses storage devices currently available in common clinical setups.

Even more critical, concerning the size of volumetric data-sets, and compared to storage problems, is visualization over the Internet. Web applications like remote diagnosis, for example, suffer from low transmission rates, even over local networks. In general, client-server solutions in the field of visualization usually are classified by the point, at which the visualization pipeline [8] is cut into a server-part and a client-part. Doing most of the visualization job at the client, for example, usually is referred to being a fat-client solution [10]. Thin clients, on the other hand, just display results of the visualization process, namely images, which entirely have been computed at the server beforehand. The trade-off between thin- and fat-client solutions is driven by the fact, that cutting the visualization pipeline at an earlier stage (fat-client solution) allows for more flexibility at the client's side (without any need to reload data). However, this advantage is gained at the expense of large-sized (volumetric) data to be downloaded,

whenever necessary (initial download, changes to parameters of the preprocess). Respectively, thin clients deal with smaller data – just result images, for example – but need to download new data, whenever any of the parameters, even just viewing parameters, are changed.

The applicability of the more flexible fat-client solution to volume visualization strongly depends on the effectivity of the compression techniques used for transmission of the data-set. Lossless compression techniques – general purpose [7] as well as volumetric data specific [6] – usually achieve rather low compression ratios (around 2), which is not sufficient to significantly widen the bandwidth bottleneck. Using lossy compression [17, 2, 12] ratios in the range of 5 to 50 can be achieved while maintaining acceptable quality of the visualization results. On the other hand, medical applications, for example, prohibit changes to the accuracy of the data, as induced by lossy compression methods. Hierarchical methods, like wavelet compression [12] combine advantages of lossy and lossless compression. By transmitting and considering just a small fraction of the coefficients (around 5%) images of acceptable quality can be generated, data values of the original volume can be reconstructed if all coefficients are considered. A useful property of wavelet compression and many lossy compression techniques is the ability to render compressed data directly, without prior expansion and decompression.

Polygonal representations of structures within the volume (e.g. of iso-surfaces) can be used to realize solutions which are compromises between a pure thin and fat client approach. The volume is maintained at the server, just the polygonal model is transmitted and rendered at the client. Changes of viewing parameters require local rendering only, just changes affecting the shape of the model require a recomputation at the server and transmission of surface data over the network. To reduce the bandwidth required to transmit the model and to improve the interactivity of rendering at low-end clients, progressive refinement as well as focus and context techniques can be used [5], trading quality of representation (in less relevant regions of the volume) for speed. A combination of server-side and client-side approaches for direct volume rendering has been presented by Engel et al. [4]. They transmit a sub-sampled volume to the client and use it for local rendering during interactions. The original volume at the server is used to create and transmit a high-quality image whenever the interaction is finished/paused.

Pure thin-client solutions on the other hand, allow to perform visualization using low-end clients making at the same time shared use of special purpose hardware at the server (multiple CPUs, VolumePro board [18] for example).

One approach to determine the effectiveness of compression techniques for volumetric data-sets and their suitability for Internet-based visualization is to compare the size of compressed volumes versus the size of images of the same data. This comparison is useful as it directly corresponds to the trade-off between thin and fat-client solutions. If sizes of compressed volume data-sets range in the same magnitude as sizes of images thereof, and given the client to provide sufficient computational performance to carry out most of the visualization steps itself, then fat-client solutions become feasible even via the Internet. In our case, we achieve compression rates such that, given a 256^3 data-set as well as 512^2 images (24 Bits per pixel) in compressed GIF-format, about 2–5 images already are bigger in size than the compressed volume data-set.
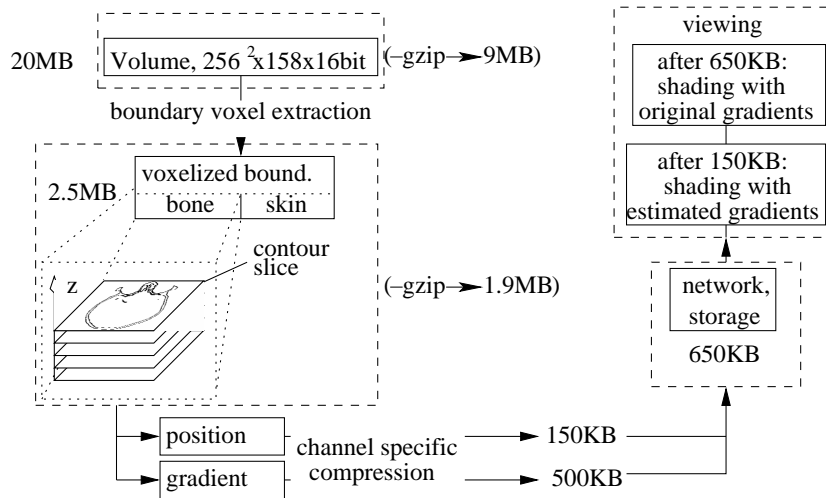
Fig. 1. Boundary extraction, compression and visualization pipeline

2 The Basic Idea

The effectivity of our approach is based on the observation that for the vast majority of applications, especially in medical visualization, volumetric data is rendered by displaying either iso-surfaces [14] or surface-like structures defined by areas of high gradient magnitude [13]. In both cases, the result of the visualization is determined by contributions of just a small fraction of all volume samples. By just coding those voxels of an object, which actually contribute to its visual appearance, the size of the data-set is greatly reduced. Thereby, a small-scale boundary representation of volumetric objects is generated (Fig. 1, Sect. 3). Compression of the boundary representation, which exploits spatial coherence among neighboring voxels, produces an extremely compact object representation (Sect. 4) which is well-suited for network transmission (Sect. 5). The information contained within this representation of objects allows interactive rendering at a client without any dependency on hardware-support, and with more flexibility regarding visualization parameters than polygonal surface representations (Sect. 6).[1]

The first step to obtain an efficient representation of bounded objects within a volumetric data-set is the identification and extraction of voxels which contribute to the object's visual representation, i.e., the boundary of the object. In our case, boundary voxels are data samples located within the object and have at least one neighboring voxel outside the object. The extraction process generates a separate boundary representation for each object within the volume. Usually 5–10% of all voxels belong to the boundary representation.

Typically, gradient information is required to evaluate a shading equation at each voxel during rendering. It is more efficient to precompute voxel gradients during bound-

---

[1] A demonstration applet is available from
   http://bandviz.cg.tuwien.ac.at/basinviz/compression/

ary extraction than to store all data values required for gradient computation at boundary voxels at rendering time.

Within our representation of an object, voxels are grouped into slices sharing the same z coordinate (See Fig. 1). Within a slice, the boundary voxels form contours of the object – a set of connected sequences of voxels. Exploiting spatial coherence of the contour, the positions of voxels within the slice are efficiently encoded into a compressed data stream. Voxel gradients are compressed in the same order as the corresponding positions, using a special compression scheme. Additional streams of voxel attributes (= data channels), like data value, gradient magnitude, etc., can be optionally encoded in a similar way. The output of the compression step is a boundary representation of volumetric objects, typically compressed by a factor of 10–100 compared to the original volume.

By transmitting the data channels in a smart order, for example, position data first, gradients last, a preview of the objects with full spatial accuracy can be displayed (appendix, Fig. 3) after transmitting just a few Kilobytes of data (using estimated gradients for shading).

The decompressed boundary representation can be rendered directly [15, 9], without prior reconstruction of a full-sized volume. Compared with a polygonal representation of the boundary surfaces, our approach preserves the full accuracy of the data-set at much lower memory cost, allows interactive rendering on low-end hardware and provides more flexibility with respect to rendering parameters. Transparency, non-photorealistic shading and the fusion with truly volumetric objects (which can be compressed using the same method) are easily possible without performance degradation.

3   Extraction of Boundary Voxels

In our approach we either use the iso-surface metaphor to specify boundary voxels, or use a predefined and explicit segmentation mask for this purpose. In the first case, voxels with a data value $\geq$ iso-value and at least one 26-connected neighbor with a value smaller than the iso-value are considered to be part of the boundary. This definition results in 6-connected sets of boundary voxels, a property useful for exploiting coherence during compression of the contours. Boundaries of objects defined using a segmentation mask, can be extracted in a similar way and also result in 6-connected sets of voxels. As voxel identification accounts only for a small part of extraction time (gradient computation is most expensive), a simple sweep method is used for this purpose. The extraction of object boundaries is performed during an interactive volume visualization session. The resulting object representation can be rendered immediately, the compressed boundary can be stored for later viewing and presentation of visualization results.

Although best compression efficiency is achieved for surface-like voxel sets, truly volumetric objects can be extracted and compressed in the same way. This is especially useful for the visualization of spatially complex structures, like vessels in medical angiography data-sets or complex chaotic attractors in the field of dynamical systems [1].
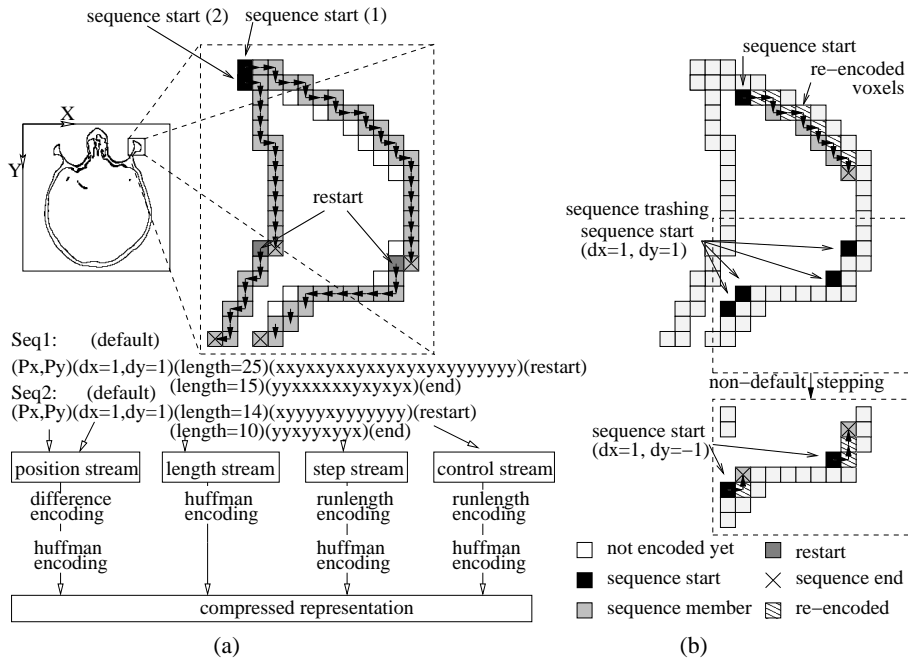
Fig. 2. Encoding of voxel positions: slice scanned from top left to bottom right a) long sequences and sequence continuations b) re-encoding of voxels and non-default stepping direction to reduce number of sequence starts and thus position specifications

For the extracted voxels, attributes (data channels) like voxel position, data value, gradient direction and magnitude, and application specific attributes are stored. When only the display of shaded surfaces is desired, storing voxel position and gradient direction is sufficient.

4 Data Compression

Individual objects within the volume are compressed separately. Voxels of each object are grouped into slices of voxels with the same z coordinate which are processed sequentially (See Fig. 1). To ensure effectivity, different data channels have to be compressed using specialized compression methods.

4.1 Position Data Channel

Boundary voxels within a single z-slice form object contours which consist of face-connected voxels (See Fig. 2). Exploiting spatial coherence and connectivity, voxels can be grouped into "*sequences*" which spatially follow the object contour. The approach is similar to "chain coding" used in binary image (text) compression [11] for contour encoding. In contrast contours of 2D objects, the boundary of a volumetric object may be thicker than a single layer of voxels, and thus requires a modified approach.

During compression, the slice is scanned for non-encoded voxels. Whenever one is found, a new sequence is started and the position of the voxel (P_x, P_y) is stored. The sequence is continued, by selecting and appending one of the neighbor voxels at it's end. As the contour voxels are face-connected, potential candidates for continuation are located at $(P_x + dx, P_y)$ or $(P_x, P_y + dy)$ with dx, dy being respectively -1 or 1. Encoding the selection of one of the four neighbors as a successor would require 2 Bits. If the choice is restricted to two neighbors by using constant values of dx and dy for a whole sequence, each voxel continuing a sequence can be specified by a single Bit, which defines whether a step by dx or dy is used. Although this restriction reduces the flexibility and thus the average length of sequences, the cost per voxel within a sequence is cut by half, outweighting the disadvantage of shorter sequences.

In cases where a direct neighbor of the trailing voxel of a sequence is present, but can not be reached using the current (fixed) dx and dy values, a *sequence restart* can be performed, continuing the sequence at this neighbor with a new value for dx or dy. To realize this, each sequence is followed by a command code which specifies whether the sequence ends, or restarts with a different stepping direction. The presence of a restart code implicitly defines the position of the start voxel of the new sequence. As the previous sequence had to be terminated, no successors of it's last voxel are present in it's dx and dy direction. One of the remaining two neighbors is the second but last voxel of the interrupted sequence, so the other one necessarily is the starting voxel of the new sequence. The new values of dx and dy are derived from dx and dy of the old sequence. Depending on whether the last step of the sequence was dx or dy either dx or dy is inverted. Although being more restrictive than with an explicit specification of dx and dy, this strategy still allows encoding of cyclic structures with a single position specification and restart commands within a *chain of sequences*.

For each combination of dx and dy values, one of the possible stepping directions is preferred, whenever both ways can be taken. The preference is chosen in a way, that a clockwise processing of closed objects will stay as close to the outer border as possible. For example, for $dx = 1$ and $dy = 1$ like in the first sequence of Fig. 2a, steps by dx are preferred.

After the creation of long sequences, usually groups of short sequences or even non-connected voxels remain. Starting a new sequence for each of these voxels is expensive. Usually most of these voxels can be encoded at a lower cost by joining them into sequences re-using voxels already encoded earlier in the process (Fig. 2b). In general, a sequence has to be continued, reusing already encoded voxels, if this allows to reach non-encoded voxels at a cost which is lower than a "sequence end" and the start of a new sequence.

As the scan for non-encoded voxels within a slice is performed in ascending x and y direction, using $dx = 1$ and $dy = 1$ as a default stepping direction for newly started sequences is usually a good solution – voxels with smaller x and y coordinates compared to the current one are already encoded in this case. In some cases however, keeping $dx = 1$ and $dy = 1$ as default directions tends to generate a lot of short sequences "sequence trashing" (Fig. 2b). Instead it is better to first search "backwards" (using $dx = -1$, $dy = 1$) and to start the new sequence using $dx = 1$ and $dy = -1$ at the last voxel found (for reasons of simplicity no backward scan was performed for the

sequences of Fig. 2a). At each sequence start 1 Bit is used to store, whether the default stepping direction $(1, 1)$, or the direction of the backward scan $(1, -1)$ is used.

For further compression, the sequence data is separated into four streams. The *position stream* stores starting positions and stepping directions. Positions are stored using Huffman encoded differences between successive coordinate values (Typically 12 Bits per starting code). The *length stream* stores information about sequence lengths (Huffman encoded, 5 Bits per sequence). The *step stream* stores the information for building up sequences (1 Bit per voxel). As dx and dy steps tend to cluster due to the presence of a preferred stepping direction, this information is run-length encoded, using again Huffman encoding for the run-lengths. The *control stream* is used for the sequence control information (end/restart, 1 Bit per sequence). As many restarts at the beginning of encoding a slice are followed by short sequences collecting isolated voxels towards the end of encoding, which leads to clustering of restart and end commands, run-length encoding combined with Huffman encoding is also used here. Combining all those streams, an average of 2 Bits is required to encode the position of a single voxel.

Within all other data channels, voxels are encoded in the same order as their position data. This ordering allows to exploit spatial coherence within voxel sequences also for attribute encoding. For subsequent occurrences of re-encoded voxels, no attribute information is stored.

4.2 Gradient Direction Channel

As a first step, gradient vectors are normalized, transformed to polar coordinates and quantized to 2x6 Bits. This gradient representation is also used by our rendering algorithm for interactive shading. By exploiting spatial coherence within the encoded stream of voxels the gradient information is reduced to 3–8 Bit per voxel, depending on the smoothness of the boundary. Both polar coordinates are encoded into separate streams, storing differences between coordinates of successive voxels. As most of the difference data consists of sequences of values in the range of $[-1, 1]$ which are occasionally interrupted by larger values or clusters thereof, the encoder switches between two different coding schemes. The first scheme is used to encode sequences of differences in the range of $[-1, 1]$ using 1 Bit for 0 (most common), 2 and 3 Bits for -1 and 1, and a 3 Bit code to switch to the other encoding scheme. Larger differences are encoded using Huffman coding with an extra symbol to switch to the encoding scheme for small values. A switch to the code for small values is only performed to encode sufficiently long sequences of small values (cost of switching).

The use of prediction techniques for estimating gradients and the encoding of the prediction error instead of encoding gradient differences seems to promise good results at the first glance. Nevertheless, tests performed using linear regression [16] with a diameter of 3 and 5 for gradient estimation, indicate that compression rates obtained using this technique are worse than our current approach.

4.3 Other Data Channels

Additional data channels, like gradient magnitude, data value, etc., are compressed in the same order as the positions of the voxels to exploit spatial coherency also. Huffman

encoding of differences of successive values and additional `zlib` compression (for further reduction of uniform areas) is used.

5 Data Transmission and Decompression

The compressed data-set consists of two parts: a header, which contains control-information about the objects and their position within the data, information about additional data channels and how to use them for rendering. The body contains voxel positions and other data channels for all objects. The data within the body is arranged in a way which allows to obtain a view on the data as early as possible during loading. Objects and data channels which are more significant for the preset visualization mappings are stored and transferred earlier than less significant data. Data channels are subdivided into blocks of a few Kilobytes each. As soon as an entire block has arrived, it can be decompressed and displayed while the following data is arriving. This allows voxel data to be rapidly updated, without having to wait for the arrival of the entire channel. Finally, as gradient information usually accounts for most of the data to be transmitted (See table 1), for boundary objects a locally computed gradient approximation (linear regression [16] with a filter size of 5 while interpreting the data as a binary object) can be displayed before the original gradient data arrives (appendix, Fig. 3). For inherently binary objects, like basins of attraction within the phase space of a dynamical system [1] the locally computed gradients can entirely replace the transmission of gradients, significantly decreasing the amount of transmitted data.

6 Rendering

In our test application, tendering of the data is performed by a Java applet at the client. A fast shear-warp-based method previously described by the Authors [15, 9] is used and extended to provide more flexible influence of data channels on the results of rendering. The 12 Bit gradient representation is used to directly index a look-up table containing shading information for interactive lighting (appendix, Fig. 4a). Using a shading table filled according to a non-photorealistic shading equation [3] and using the result to modulate voxel opacity, interactive non-photorealistic rendering can be realized. Using gradient-based shading and an additional gradient magnitude channel, the classical gradient-modulated transfer functions of Levoy [13] can be realized. Using one additional data channel (containing distance information) to modulate either color or opacity (appendix, Fig. 4b) the visualization of contacts between objects can be enhanced.

7 Results

Table 1 presents the compression rates obtained by applying our technique to a collection of data-sets from different application fields. The head and hand data-sets are CT scans containing objects typical for medical applications. Bone and skin surfaces extracted from the data are usually made up from 1–4% of all voxels. Using our compression scheme the boundary data is compressed by a factor of 20–90 compared to

data-set	volume size	obj. voxels	Bit/pos	Bit/grad	Bit/voxel	file(w/o grad.)	ratio to gzipped vol.
head-bone	$256^2 * 158$	378k	2.0	7.0	9.0	430k(95k)	1:22(1:97)
head-skin	$256^2 * 158$	231k	2.1	5.8	7.9	229k(60k)	1:40(1:154)
hand-bone	$256^2 * 232$	191k	2.5	7.8	10.3	246k(60k)	1:45(1:186)
hand-skin	$256^2 * 232$	170k	2.0	4.0	6.0	126k(41k)	1:89(1:273)
engine	$256^2 * 110$	298k	1.7	5.1	6.8	253k(64k)	1:13(1:51)
teapot	256^3	152k	1.7	3.4	5.1	80k(28k)	1:4(1:11)
attractor	256^3	769k	1.8	4.9*	6.7	639k(170k)	$-$**
basin	256^3	292k	2.2	0.6*	2.8	104k(80k)	$-$**

Table 1. Compression survey. * Scalar value channel instead of gradients. ** The attractor and basin data-sets have been extracted from a volume with a vector of several scalar values at each voxel directly within the simulation application. No volumetric representation was available.

the original volume when compressed with `gzip`. If gradient information is not stored but approximated at the client the compression factor increases to 100–270. The cost of compressing voxel positions within such data-sets is relatively independent of the surface shape 2–2.5 Bit/voxel. The cost for storing gradients depends on the smoothness and curvature of the surface and varies between 4 and 8 Bit/voxel. For objects with artificial, "well-behaved" surfaces like the CT scan of an engine block or the voxelized teapot, better compression is achieved for both voxel position and gradient data. The attractor and basin of attraction data, obtained from a simulation of a dynamical system, is also effectively compressed – especially as the basin boundary is derived from a binary classification of space and no gradient information has to be stored – it can be reconstructed from the surface shape at the client. Compression for each of the examples mentioned above takes approximately one second on a PIII/733 PC. Decompression timings for locally stored data are similar on the same PC. An applet which implements the techniques described in this paper and all compressed data-sets discussed and depicted here are available at `http://bandviz.cg.tuwien.ac.at/basinviz/compression/`.

8 Conclusions

Many applications of volume visualization require the display of objects boundaries. Using our compact volume representation, volume visualization becomes feasible even over the Internet, while still providing full spatial accuracy. Representing just the boundary voxels of objects reduces the amount of data to be transmitted or stored dramatically. By exploiting known properties of the boundary voxels (like spatial coherence and intervoxel connectivity) the data is further compressed. The resulting data representation is smaller by a factor of 20-250 than the volume compressed with `gzip`. The location of voxels within the volume is compressed very efficiently to about 2 Bit/voxel. The compression rates for gradient data are lower, in the range of 3-8 Bit/voxel, as gradient data is derivative information compared to the original data, containing less spatial coherence. Using a proper gradient reconstruction scheme, gradients can be estimated from voxel positions only, allowing to display objects just after the well-compressed position data has arrived, instead of waiting for the original gradient information. The display of

the boundary data can be performed in pure software (Java) at interactive frame rates without the need for any hardware support .

Acknowledgements

References

1. G.-I. Bischi, L. Mroz, and H. Hauser. Studying basin bifurcations in nonlinear triopoly games by using 3D visualization. *accepted for publication in Journal of Nonlinear Analysis*.
2. T. Chiueh, C. Yang, T. He, H. Pfister, and A. Kaufman. Integrated volume compression and visualization. In *Proceedings IEEE Visualization '97*, pages 329–336, 1997.
3. D. Ebert and P. Rheingans. Volume illustration: non-photographic rendering of volume models. In *Proceedings IEEE Visualization 2000*, pages 195–202, 2000.
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. In *Proceedings IEEE Visualization 2000*, pages 449–452, 2000.
5. K. Engel, R. Westermann, and T. Ertl. Isosurface extraction techniques for web-based volume visualization. In *Proceedings IEEE Visualization '99*, pages 139–146, 1999.
6. J. Fowler and R. Yagel. Lossless compression of volume data. In *Proceedings IEEE Volume Visualization Symposium '94*, pages 43–50, 1994.
7. J. Gailly and M. Adler. gzip. URL: `http://www.gzip.org`.
8. R. Haber and D. McNabb. *Visualization idioms: A conceptual model for scientific visualization systems, visualization in scientific computing*, pages 74–93. 1996.
9. H. Hauser, L. Mroz, G.-I. Bischi, and E. Gröller. Two-level volume rendering - fusing MIP and DVR. In *Proceedings IEEE Visualization 2000*, pages 211–218, 2000.
10. M. Jern. Information drill-down using web tools. In *Proceedings of the 8th EUROGRAPHICS Workshop on Visualization in Scientific Computing*, pages 1–12, 1997.
11. R. Estes JR and V. Algazi. Efficient error free chain coding of binary documents. In *Proceedings of the Data Compression Conference*, pages 122–132, 1995.
12. C. Kurmann L. Lippert, M. Gross. Compression domain volume rendering for distributed environments. In *Proceedings of Eurographics '97*, pages C95–C107, 1997.
13. M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8(5):29–37, May 1988.
14. W. Lorensen and H. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of ACM SIGGRAPH '87*, pages 163–189, 1987.
15. L. Mroz, R. Wegenkittl, and E. Gröller. Mastering interactive surface rendering for java-based diagnostic applications. In *Proceedings IEEE Visualization 2000*, pages 437–440.
16. L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4D linear regression. In *Proceedings of Eurographics 2000*, pages C–351–C–357.
17. P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *Proceedings IEEE Visualization '93*, pages 11–18, 1993.
18. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *Proceedings of ACM SIGGRAPH '99*, pages 251–260, 1999.
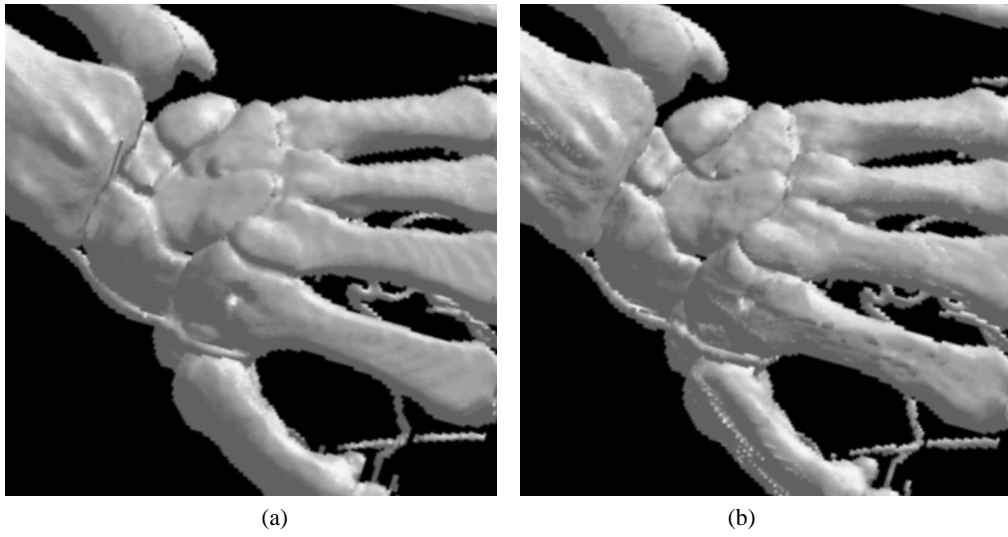
Fig. 3. Estimated gradients (a) are used for shading until the original gradient data has arrived (b)
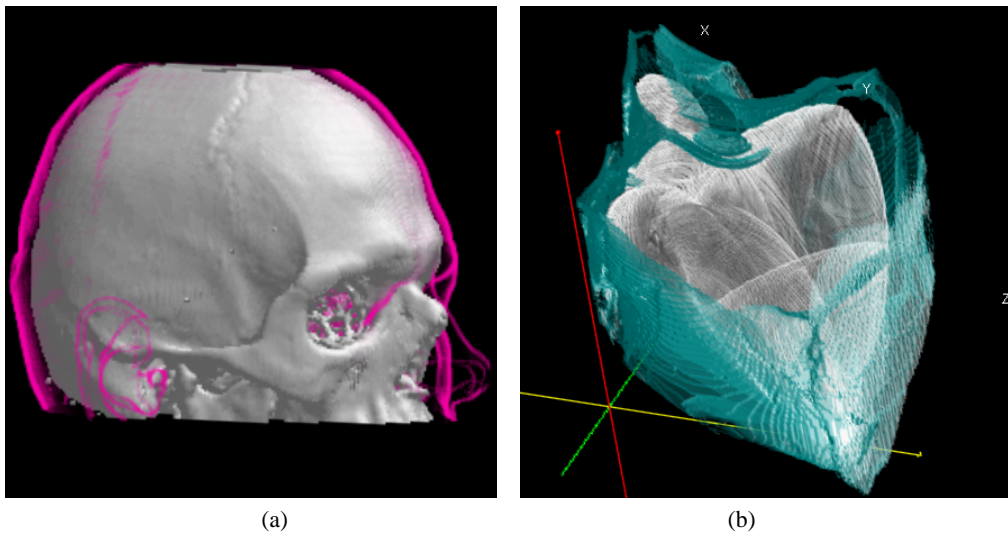

Fig. 4. a) By adjusting the visualization mappings at the client the skin surface has been rendered using a non-photorealistic technique over the conventionally shaded skull. b) A data channel containing distance information has been used to modulate opacity of the basin surface to emphasize areas of almost-contact between the surface and the attractor contained within.