Hardware Accelerated Rendering of Emissive Volumes

Wolfgang Heidrich

University of Erlangen, IMMD IX, Computer Graphics Group
Am Weichselgarten 9, D-91058 Erlangen, Germany
Email: heidrich@informatik.uni-erlangen.de

Abstract

Traditional volume rendering is computationally expensive and can therefore only be performed at low frame rates. In recent years algorithms have been developed that are specifically tailored towards the use of graphics hardware for volume rendering. On systems equipped with the appropriate hardware, these algorithms are capable of rendering volumes at interactive rates.

Unfortunately, existing algorithms rely on advanced graphics features like texture mapping, which are not available on contemporary low-end systems. In this paper we describe a novel approach for the hardware based rendering of emissive volumes, which only requires very basic support by the graphics hardware, such as flat shaded polygons, depth cueing and depth buffer. Therefore the algorithm is capable of exploiting low-end graphics hardware without support for texture mapping and other advanced features.

1 Introduction

Volume rendering is concerned with the generation of images that reveal the internal structures of volume data sets obtained by simulations or measurements. Volume rendering achieves this goal by interpreting the values in the data set as coefficients for the emission ϕ and absorption μ of light at a given point in 3-D space. These data points are treated as discrete samples of continuous volumetric functions $\phi(\vec{x})$ and $\mu(\vec{x})$. The data set can therefore be rendered by tracing rays from a given camera position through the volume, and integrating the intensities along these rays. This is done using the *emission/absorption integral* [5]:

$$\Phi[d] = \int_0^d e^{-\int_z^d \mu[\zeta]d\zeta} \phi[z]dz + e^{-\int_0^d \mu[z]dz} \Phi[0]. \tag{1}$$

The value $\Phi[d]$ is the flux density reaching the image plane at $z = d$, and $\Phi[0]$ defines a background intensity at $z = 0$. $\mu(\vec{x})$ denotes the linear absorption coefficient of the medium, while $\phi(\vec{x})$ represents the flux density per unit length emitted towards the image plane by suspended particles.

By manipulating the way in which the values of the original volume data are interpreted as absorptive $\mu(\vec{x})$ and emissive $\phi(\vec{x})$ properties, various effects can be achieved, including iso-surfaces [7, 8] and apparently opaque objects.

1.1 Emissive Volumes

Often, the data sets to be visualized only contain a single scalar value for every point in the volume. This is for example true for most medical data sets, like MRA or ultrasound data. In this case, it is natural to directly map this scalar value to either the absorption $\mu(\vec{x})$ or the emission $\phi(\vec{x})$, respectively, and to set the other coefficient to zero. In the latter case, where $\mu(\vec{x}) = 0$, the volume is called *purely emissive* and the integral in (1) reduces to

$$\Phi[d] = \int_0^d \phi[z]\, dz + \Phi[0], \tag{2}$$

which is linear with respect to $\phi[z]$.

Rendering of emissive volumes plays an important role in medical imaging and other areas of visualization. It is based on the metaphor of emissive particles suspended in a transparent medium. Given such a volume of emission values, the rendering task is to project the 3-D volume onto a 2-D image plane. This projection maps a single brightness value to each pixel on the image plane, depending on the emission of the voxels located on the projection ray.

The most intuitive and straight-forward projection operation is the so-called "summation

rendering". For every pixel in the image, it directly evaluates the integral from Equation 2 over all the voxels along the projection ray corresponding to this pixel.

Another commonly used projection operator is "maximum projection rendering", which assigns to every pixel the maximum emission of all voxels along the projection ray corresponding to this pixel. This is characterized by the following projection formula:

$$\Phi[d] = \max(\max_{z \in (0,d]} (\zeta \phi[z]), \Phi[0]).$$

The scale factor ζ converts the flux density per unit length $\phi[z]$ to a flux density Φ.

Maximum projection rendering is particularly useful for any application in which a strong signal is produced over a small volume, and might be obscured by low-level background noise produced over a much larger volume.

In the following we first review and discuss a recent algorithm for hardware accelerated rendering of volumes based on texture mapping. Then we describe a novel approach which only uses flat shaded polygons to render the volumes, and thus overcomes the performance penalties of texture mapping on low-end machines. Finally we show how volumes and surfaces can be combined into one image with our approach, and conclude with some results and performance measurements.

1.2 Hardware Accelerated Volume Rendering: State of the Art

Existing approaches to hardware accelerated volume rendering use texture mapping to sample the volume at discrete slices and then construct an image by computing a weighted sum of those slices. Early methods used only 2-dimensional textures that represented slices of the original volume [2], while more recent approaches treat the whole volume as a 3-dimensional texture.

The volume is rendered by intersecting the volume with a set of planes parallel to the viewing plane. This is simply done by loading the volume data into texture memory, and drawing the planes as polygons, with the texture coordinates set up correctly (see [3] and [6] for details). In order to generate the final image, the slices obtained this way are then blended together using alpha blending.

On high-end machines supporting hardware texture mapping, this approach to volume rendering can be extremely fast. The rendering performance only depends on the number of slices blended together, and not on the size of the volume, as long as the volume completely fits into the texture memory. If, however, the size of this memory is exceeded, the volume has to be broken down into smaller sub-volumes, called "bricks", and these have to be swapped in and out of the texture memory for every frame. As a consequence, the rendering performance will usually be limited by the bandwidth of the bus used to transfer the bricks into texture memory.

On low-end machines without hardware texture mapping support, the performance penalties for this approach are even more striking. Usually bus bandwidth is not a problem on these platforms, since the volume resides in main memory and thus does not have to be transferred over a bus continuously. However, the CPU load imposed by rendering several hundred large texture-mapped polygons with tri-linear interpolation is so high, that even on the most modern CPUs the frame rate degrades to one frame every few seconds.

In this paper we describe a novel approach for volume rendering, which is purely based on rendering flat shaded polygons without texture mapping, and thus scales well with both the size of the data set and the available graphics hardware.

2 A Discretization of Emissive Volumes

As a first step for rendering a volume data set, we discretize the continuous function it represents. A discrete version of the volume can be derived by making a transition from a continuous emission function $\phi(\vec{x})$ to a set of discrete emission levels $\{\phi_1, \phi_2, \ldots, \phi_n\}$. The iso-surfaces corresponding to these levels can be seen as thresholds defining a number of nested volumes or "shells" [13].

Using this discretization, a volume renderer based on ray tracing could be implemented by testing every projection ray for intersection with these iso-surfaces. For summation rendering the brightness values for all intersected shells would then simply be added together, while for maximum projection rendering the emission ϕ_i of the brightest intersecting shell i would be taken.

The advantage of this approach is that, once the volume is discretized, ray tracing no longer needs to be used to render the image. Instead, as we shall see below, we can use conventional 3-D graphics hardware to render a polygonal model of the shells-surfaces. The iso-surfaces corresponding to these shells have to be generated using one of the well-known algorithms, such as marching cubes [9].

2.1 Summation Rendering

It is straight-forward to develop a hardware-accelerated version of the discretized summation rendering algorithm as described above. Using a standard hardware model, like the one of the OpenGL library [10], the extracted iso-surfaces are simply drawn with per-fragment blending set up in such a way, that the intensities for every pixel are added up in the frame buffer (instead of letting the new color value replace the old contents of the frame buffer).

In cases where such a blending mode is not available (it has only recently been added to the version 1.1 of the OpenGL API [11], which is not yet widely available), a similar effect can be obtained by rendering the shells into separate images and adding those using an accumulation buffer.

Unfortunately, both blending and accumulation buffer operations are usually not supported in hardware on low-end systems, so that the performance of these operations is relatively low (see also Section 5).

2.2 Maximum Projection Rendering

The situation for maximum projection rendering is somewhat more complicated. Like for summation rendering the polygons of the iso-surfaces can be projected onto the image plane using simple polygon rendering. However, instead of summing up the intensities in the frame buffer, it is now necessary to perform a maximum operation on the intensities. For every rendered pixel the new intensity has to be compared to the value that already resides in the frame buffer, and only if the new value is brighter it replaces the older one. Such an operation is neither directly supported by the most contemporary graphics hardware, nor by graphics libraries such as OpenGL.

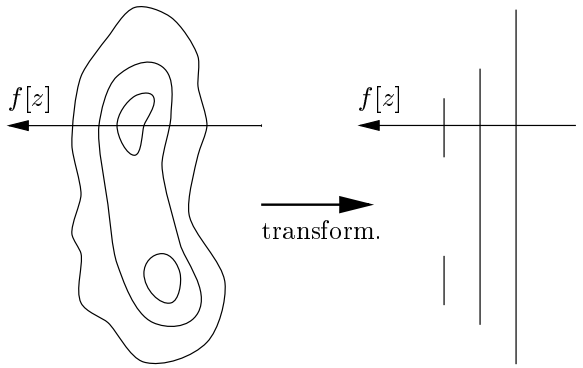


Figure 1: The result of the geometric transformation for maximum rendering: the z component of each point is set to the emission of the iso-surface it belongs to

However, using a simple geometric transformation, it is possible to exploit conventional z-buffer hardware to implement the maximum operator. Maximum projection rendering as described above is equivalent to an orthographic or perspective transformation followed by a maximum operation. Such a maximum operation is already provided by the z-buffer algorithm during hidden surface removal. We would like to use this z-buffer for our maximum operation on intensities as well. Therefore we have to come up with a geometric transformation which reorganizes the geometry so that brighter points are closer to the image plane than darker points.

A simple way to accomplish this, if the z-axis is perpendicular to the image plane, is to set the z component of each point to its emission after the projective transformation. The transformation matrix which does this is

$$\begin{bmatrix} x \\ y \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \phi_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \qquad (3)$$

where ϕ_i is the emission of the iso-surface to which the point belongs. This transformation can be implemented as a scaling by factor 0 in z direction followed by a translation by ϕ_i. The result of this transformation is shown in Figure 1.

Note that, if a perspective transformation is used, the above transformation has to be executed *after* the perspective is done. Otherwise perspective foreshortening will change the proportions of the geometry: since brighter surfaces

are moved closer to the eye, the perspective transformation would make them appear larger than dark ones. Applying the above transformation after the perspective ensures that the size of the projected objects on the image plane does not change.

3 Depth Cueing

It is often useful to add additional depth cues to the image in order to improve the 3-dimensional impression. One of the simplest depth cues to implement is an intensity ramp that dims parts of the volume farther from the eye. Such a linear ramp is supported by most of today's 3-D graphics accelerators, and is usually called *linear fog*. A linear ramp along a projector can be described as

$$\beta[z] = \beta_0 + \beta_m z,$$

with $\beta_0 \geq 0$ and $\beta_m > 0$. The value β_0 is the attenuation at the background image, and β_m is the rate of change of this attenuation per unit length along the projector.

For depth cueing in volume data sets, this linear ramp can be directly applied to the algorithm for summation rendering we described in Section 2.1 above. Obviously this is not possible for the maximum projection algorithm from Section 2.2, because the transformation from Equation 3 throws away all of the original depth information, and replaces it by the brightness value of the shells.

In order to add linear depth cueing to maximum projection images, we therefore have to add some of the depth information back in. We do so by applying the maximum operator to a brightness that is multiplied by the linear depth ramp:

$$\Phi[d] = \max(\max_i(\zeta\beta[z_i]\phi_i), \beta_0\Phi[0]) \qquad (4)$$

In order to control the degrees of freedom the depth ramp provides, we have the choice to either directly adjust its parameters β_0 and β_m, or to fix those parameters and apply a linear function to the z component of each point instead.

As we shall see below, it is convenient to choose the latter approach by applying the depth cue to a shifted and scaled $z_i' = az_i + b$. Under the assumption that background is darker than the brightest point along a projector, Equation (4)

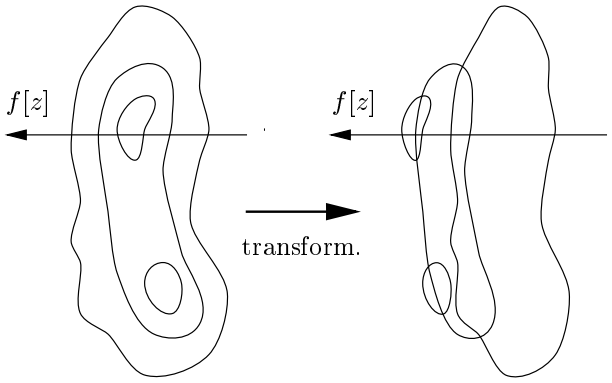


Figure 2: The result of the geometric transformation including depth cueing: the geometry is scaled and translated so that brighter points are closer to the image plane.

reduces to

$$\Phi[d] = \max_i \left(\zeta(\beta_0 + \beta_m(az_i + b))\phi_i\right),$$

which can be rearranged to

$$\Phi[d] = \max_i \left(\zeta\beta_m \left(\phi_i a z_i + \phi_i(b + \frac{\beta_0}{\beta_m})\right)\right).$$

In other words, the transformation to control the linear ramp can be pushed into the transformation we do before the maximum operation, and only a single maximum comparison is needed on the combined depth/intensity. Setting

$$\begin{array}{rcll} a_i^* & = & \phi_i a & \textit{and} \\ b_i^* & = & \phi_i(b + \beta_0/\beta_m), & \end{array}$$

transformation (3) can therefore be redefined as

$$\begin{bmatrix} x \\ y \\ z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a_i^* & b_i^* \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \qquad (5)$$

Figure 2 demonstrates the effect of this transformation.

Instead of drawing each iso-surface with a color corresponding to its emission, like in Section 2.2, we now simply transform each iso-surface with the appropriate matrix, and then draw it in full brightness. The hardware depth cueing of the graphics system we use will take care that surfaces with higher emission will appear brighter on the screen than ones with lower emission.

We now have parameters β_0, a and b to customize the rendering algorithm. If we set $a = 0$, $b = 1$, and the brightness of the background image $\beta_0 = 0$, we have exactly the algorithm described in Section 2.2, without any depth cueing. By varying b, we can adjust the contrast between iso-surfaces of different brightness.

The parameter a controls the amount of depth cueing added to the image. It is actually the ratio of a and b that gives the full tradeoff of depth cueing to intensity. Increasing β_0 will increase the overall brightness of the image.

4 Combining Volumes and Surfaces

Although there are applications where volumes are the only objects that have to be displayed, there are also occasions where volumes and surfaces have to be combined in one image. Consider, for example, the concrete application of planning or simulating a surgery. The tools used in the surgery, as well as additional objects like screws that remain within the body of the patient have to be combined with the volume for the final image.

When combining surfaces and volumes, it is important to handle the occlusion problem correctly. That is, surfaces should occlude other surfaces as well as those parts of the volume that are further away. The projection operator for volume rendering should be applied to only those parts of the volume that are visible from the eye point.

For summation rendering as described in Section 2.1, this is easily achieved by rendering the surfaces first, with depth buffering enabled. Then writing to the depth buffer is disabled but the depth comparison itself is kept active (in OpenGL this is done using the function `glDepthMask`, see [1] for details). With this setting, all the shells are rendered as described in Section 2.1. In pseudo-code the complete algorithm results to:

```
render surfaces
disable writing to depth buffer
set up blending
render all shells
```

The situation for maximum rendering is more difficult. Since the depth information for the shells is modified according to Equation 5, simple depth-buffered rendering of the surfaces is not sufficient. Instead, in order to handle occlusion correctly, the surfaces in the scene have to be rendered multiple times, once for every shell. Since the shells of the volume typically contain several orders of magnitude more polygons than the surfaces themselves, this usually does not impose a severe performance penalty.

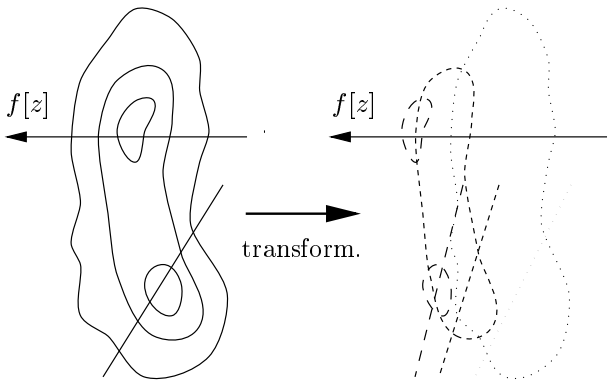


Figure 3: Combining volumes and surfaces. The polygonal objects are rendered into the depth buffer multiple times, once for every shell

In detail, the algorithm works as follows (also see Figure 3). First the surfaces are rendered with depth buffering enabled. Then the depth buffer is cleared. Starting with the dimmest iso-surface, the following steps are then performed for each shell: first, the transformation from Equation 5 is pushed on the matrix stack. Then writing to the color buffer is disabled, and the surfaces are rendered into the depth buffer using the current transformation. Afterwards, the shell is rendered with writing to the color buffer turned back on. Because at this point the depth buffer contains the distances of the surfaces under the current transformation, the occlusion problem is solved correctly. The pseudo-code of this algorithm is shown below.

```
render polygonal objects
clear depth buffer
for each shell (in order of
                increasing brightness)
  set up transformation (5)
  disable rendering to color buffer
  render polygonal objects
  enable rendering to color buffer
  render shell
end
```

5 Results

A volume visualization program for regularly gridded data has been implemented using the methods described in this paper. The program uses the 3D graphics library OpenGL [1] to exploit the graphics hardware of different platforms in a portable way. It has been tested on workstations from Silicon Graphics (SGI) and Digital Equipment (DEC).

The user interface allows for three different rendering modes: maximum rendering, summation rendering, and the usual lighted rendering of iso-surfaces. In the latter mode only the outermost shell will be visible, of course. Three scrollbars provide the user with full control over the rendering parameters β_0, a and b.

The program also allows the user to interactively add or delete iso-surfaces, or to temporarily disable individual surfaces. For the creation of the iso-surfaces a simple marching cubes algorithm [9] is used.

The resulting volume model can be interactively rotated on the screen, where the performance depends on the complexity of the volume data, the number of iso-surfaces, and the graphics hardware.

Several tests with different data sets were made in order to measure the performance on different graphics platforms. For the first measurement we chose the $64 \times 64 \times 64$ data set HIPIP from the Chapel Hill Volume Rendering Test Data Sets (CHVRTD), and created 11 iso-surfaces with 44846 polygons. A second measurement was done with the same data, but with only 6 iso-surfaces and 14844 polygons.

Another data set (SAT) contains a $21 \times 21 \times 51$ grid of saturation values from a groundwater simulation of oil contamination [12]. From this data 20 iso-surfaces with a total of 16846 polygons were created. Summation renderings with and without depth cueing are shown in the bottom two pictures of Figure 4.

Finally, an MR angiogram of a human brain was used. This data, which can be obtained from the UMDS Image Processing Group, consists of a $256 \times 256 \times 124$ grid, from which 12 iso-surfaces with 177594 polygons have been created. Maximum projection rendering is especially useful for angiograms, because they contain a lot of localized detail which would be obscured by other rendering methods. The top four images of Figure 4 show the MR angiogram data set rendered with different parameter settings. The two images on the right are combined renderings of a volume with a single polygon. This polygon can be used to clip away details at the far side of the volume, and is therefore an aid in understanding the data.

	Onyx	Indigo	Alpha
Maximum			
Hipip 1	8.51	1.59	1.63
Hipip 2	>20	4.50	4.26
Angiogram	2.66	0.74	0.39
Sat	>20	4.01	3.90
Summation			
Hipip 1	5.60	0.23	0.047
Hipip 2	12.0	0.41	0.080
Angiogram	1.38	0.071	0.040
Sat	5.48	0.056	0.028

Table 1: *Frame rates for different data sets on different platforms in frames per second. The SGI Onyx is equipped with VTX graphics, the SGI Indigo has a XS-24 board, and the DEC Alpha a PXG Turbo.*

Table 1 gives an overview of the performances measured for all the data sets on different platforms.

The figures shown for the Silicon Graphics Onyx with VTX graphics are somewhat inaccurate, because for these high frame rates the time spent synchronizing with the CRT refresh signal is significant. Since the refresh frequency of the CRT was set to 60Hz, and frame rates slightly above 20 have been measured, the rendering of one image takes between 2 and 3 refresh cycles, which corresponds to a frame rate of $20 - 30$ frames per second.

The results for all platforms show that reasonable frame rates can be achieved for maximum projection rendering, even on relatively low-end graphics acceleration boards. On the other hand the figures also show that summation rendering on low-end platforms without hardware support for blending and/or accumulation buffer is too slow to be useful in practice.

In comparison to texture based volume rendering, we have shown an algorithm that scales well with both the available hardware and the size of the volume. While we think that the texture based methods will continue to be important for the high-end, especially with hardware

texture mapping becoming more widely available, the method described in this paper might be an alternative for applications that have to run on PCs and low-end workstations.

Acknowledgments

I would like to thank Andre Unger and Nathan P. Konrad, University of Waterloo for providing the SAT data set.

The data set HIPIP originates from a simulation done by Louis Noodleman and David Case, Scripps Clinic, La Jolla, California, and can be obtained from the CHVRTD test suite at `omicron.cs.unc.edu`.

The MR angiogram data set is available from the United Medical and Dental Schools (UMDS) Image Processing Group in London, and can be found on the web at `www-ipg.umds.ac.uk/archive/heads.html`.

Parts of the algorithms described in this paper were developed together with Michael McCool from the University of Waterloo and John Stevens from the Toronto Western Hospital, and published in [4].

References

[1] OpenGL ArchitectureReview Board. *OpenGL Reference Manual: The Official Reference Document for OpenGL.*

[2] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *ACM Computer Graphics (ACM SIGGRAPH '88 Proceedings)*, 22(4):65–74, August 1988.

[3] Robert Fraser. Interactive Volume Rendering Using Advanced Graphics Architectures. Technical report, Silicon Graphics Computer Systems, 1995.

[4] Wolfgang Heidrich, Michael McCool, and John Stevens. Interactive maximum projection volume rendering. In *Proc. IEEE Visualization '95*, pages 11 – 18. IEEE Computer Society Press, 1995.

[5] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *ACM Computer Graphics (ACM SIGGRAPH '89 Proceedings)*, 23(3):271–280, July 1989.

[6] Todd Kulick. Building an OpenGL Renderer. *SGI Developer News*, 1996.

[7] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

[8] Marc Levoy. *Display of Surfaces from Volume Data.* PhD thesis, University of North Carolina at Chapel Hill, Department of Computer Science, May 1989.

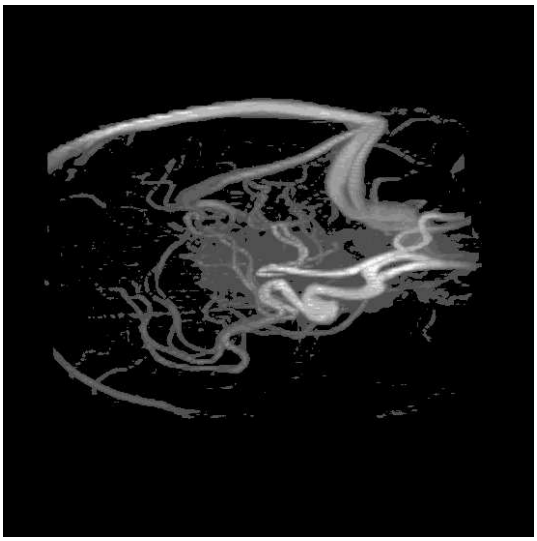
[9] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Computer Graphics (ACM SIGGRAPH '87 Proceedings)*, 21(4):163–189, July 1987.

[10] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL.* Addison-Wesley, 1993.

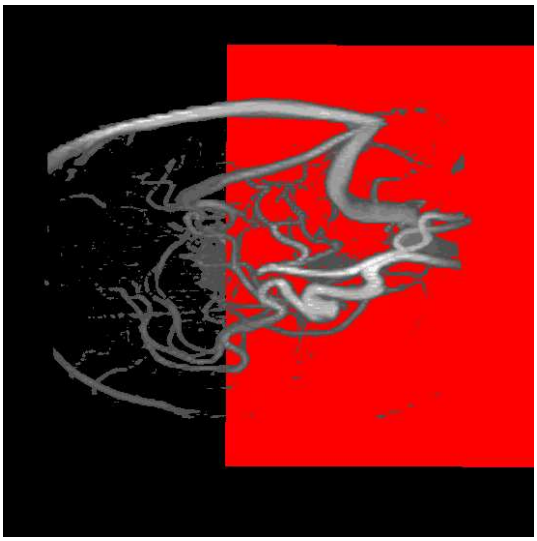
[11] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.1). Technical report, Silicon Graphics Inc., 1995.

[12] A.J.A. Unger, E.A. Sudicky, and P.A. Forsyth. Mechanisms controlling air sparging for remediation of heteogeneous formations contaminated by dense non-aqueous phase liquids. In *Water Resources Research*, 1995. accepted for publication.

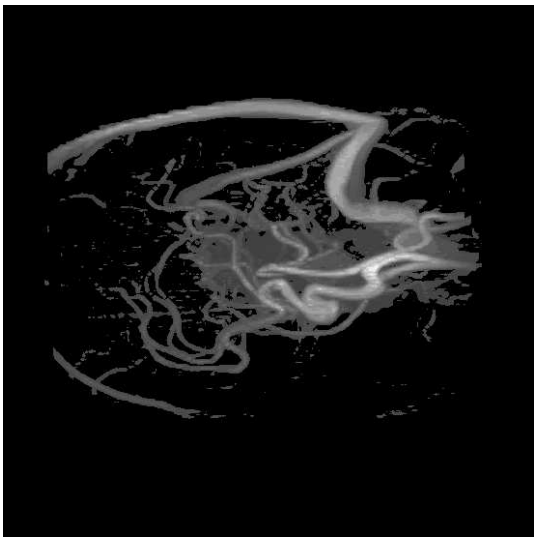
[13] J. Upda and D. Odhner. Shell rendering: Fast volume rendering and analysis of fuzzy surfaces. *Journal of Digital Imaging*, 4:159–168, 1991.



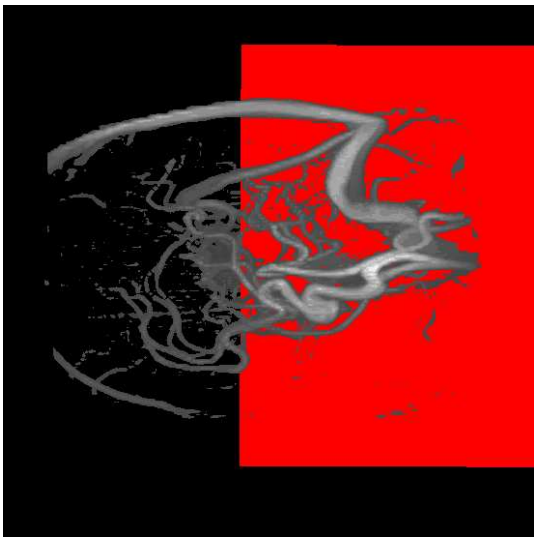
MR data, max. rendering



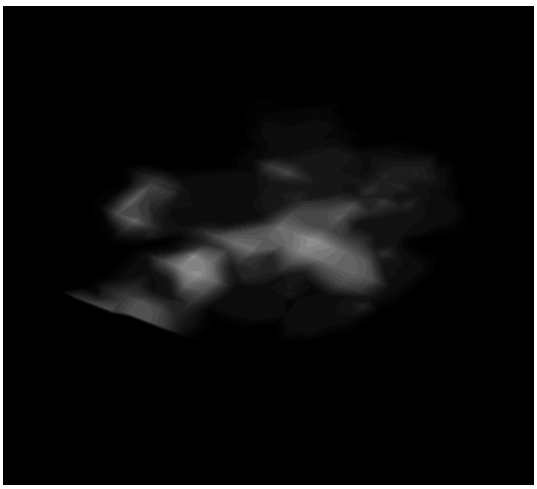
Max. rendering, with surface



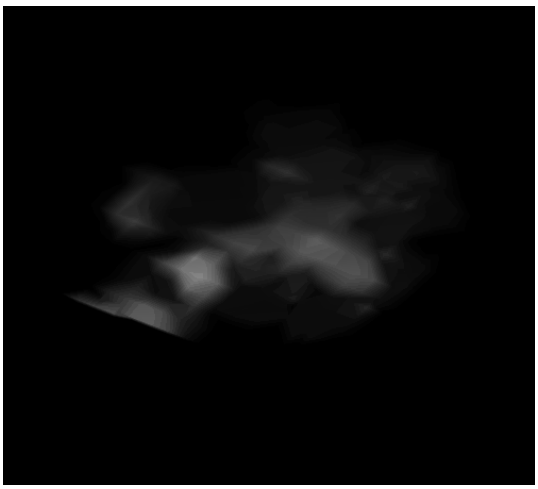
Max. rendering, depth cued



Max. rendering, depth cued, with surface



Groundwater data, sum. rendering



Sum. rendering, depth cued

Figure 4: Several sample images rendered with the algorithms presented in this paper. Please refer to the color section for a colored version.