Interactive Volume Illustration and Feature Halos

Nikolai A. Svakhine
Purdue University
svakhine@purdue.edu

David S.Ebert [*]
Purdue University
ebertd@purdue.edu

Abstract

Volume illustration is a developing trend in volume visualization, focused on conveying volume information effectively by enhancing interesting features of the volume and omitting insignificant data. However, the calculations involved have limited the illustration process to non-interactive rendering. We have developed a new interactive volume illustration system (IVIS) that harnesses the power of programmable graphics processors, and includes a novel approach for feature halo enhancement. This interactive illustration system is a powerful tool for exploration and analysis of volumetric datasets.

1. Introduction

Efficient exploration and highlighting of volume structures has been a major emphasis of visualization research. The main challenge is conveying the volume structure effectively to the user. Traditionally, visual realism has been a goal of many visualization approaches. Although photo-realistic approaches can be used with volume rendering as a technique to enhance volumetric perception (e.g. Phong lighting model for isosurfaces), realism might not be the best approach for volumetric datasets. In contrast, the volume illustration approach combines traditional volume rendering with non-photorealistic (illustrative) techniques. The main goal of the volume illustration approach is to enhance the expressiveness of volume rendering by highlighting important features within a volume, providing depth cues and omitting "uninteresting" data. The resulting images resemble a technical or medical illustration, hence the term "volume illustration".

Our current work extends the volume illustration research by Ebert and Rheingans [5] to interactive rates through utilization of modern graphics hardware and new algorithms for some illustration techniques. The original work presented a set of flexible volume rendering enhancements at non-interactive rates.

This new work takes advantage of current PC graphics hardware capabilities and uses the recently introduced Cg language[1], which is a compiler for facilitating vertex/fragment programs development and making the implementation details reasonably transparent. With the help of these tools, a number of volume illustration enhancements can be implemented in a straightforward manner. However the algorithms for complicated enhancements, like feature halos, require redesign. This paper presents methods for adapting and accelerating the enhanced visualization process using modern graphics hardware.

2. Related work

Volume illustration can be viewed as non-photorealistic rendering (NPR) applied to volume visualization. The approaches for the problem can be roughly divided into two categories: the first approach enhances standard volume rendering algorithms with NPR techniques, while the second approach applies illustrative drawing styles to volumes, creating a new way to demonstrate volume features.

The first category of volume illustration techniques is described by Ebert and Rheingans [5]. Csebfalvi et al. [4] use accelerated ray-casting and shear-warp rendering for fast object contour rendering and Lum et al. [10] describe using graphics board capabilities and clusters of PCs to achieve interactive rates for tone shading, silhouettes, and depth-based color cues [11].

The second approach applies common drawing styles to volume illustration. One popular technique is hatching, or rendering images by the means of monochrome strokes. Praun [14] and Hertzmann [6] describe hatching for surface shape illustration, and Nagy and Westermann [12] describe hatching for conveying the shape of volume isosurfaces. Interrante [7] uses the approach of stroke advection along the natural "flow" over the surface of an object, achieving a similar result. Lu et al. [9] introduced another approach for non-photorealistic rendering using volumetric stippling.

[*]Purdue University, Rendering and Perceptualization Lab (PURPL),West Lafayette, Indiana, USA

3. Algorithm

We use texture-based volume rendering [2] in combination with programmable fragment programs for our volume illustration system. Some illustration techniques are easily adapted to modern graphics hardware and are briefly discussed in Section 3.2. Feature halos are computationally very expensive and, therefore, we have developed a new algorithm for rendering feature halos interactively. This algorithm is described in Section 3.3.

For volume rendering, the volume is sliced by view-aligned quadrilaterals that are rendered in back-to front order and blended together to form the final image. Extensive hardware programmability allows us to enhance the original approach, pushing the transfer function and enhancement calculation down to the fragment processing level as shown in Figure 1.

As each slice is being rendered, the application level program clips the slice against the volume boundary, while shading and enhancement occur at the fragment level. Thus to change the rendering parameters, we only need to change the input of the fragment program, which avoids reprocessing the data. Figure 1 illustrates the general operations of the rendering process for a single slice.

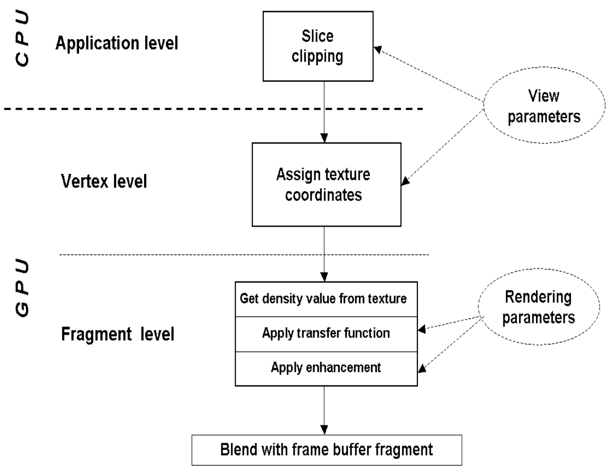


Figure 1. Interactive rendering pipeline for single slice

3.1. Fragment program structure

The fragment program reads the density value from the volume dataset texture and applies the transfer function and volumetric illustration enhancements. There are two ways that the transfer function can be applied in the fragment program. The first way is in-shader calculation, where the function takes the density value as input and uses the fragment GPU instructions to calculate the RGBA output. The second way uses a 1-D texture that stores RGBA output and is indexed by the density value.

The first method provides us with floating-point precision, since the calculation occurs inside the fragment program and does not involve 8-bit textures. However, an implementation of transfer functions with complicated profiles would require long fragment programs, which would cause slow performance. Therefore, in order to provide speed and flexibility in the transfer function specification, we use the second method. The precision is increased with new graphics hardware which provides 16-bit textures through OpenGL extensions.

3.2. Enhancements

Our basic set of illustrative enhancements are the feature and orientation enhancements introduced in [5]. Since the adaptation of most of these enhancements to graphics hardware is straightforward, they are briefly summarized below. The new feature halo enhancement algorithm is presented in Section 3.3.

Boundary enhancement: Areas with high gradient represent boundaries between materials. To highlight them, we need to increase the opacity proportional to the gradient magnitude. For a more effective outline of the high gradient areas, we use the gradient magnitude term raised to a power:

$$o_e = o_o \cdot |\overrightarrow{\nabla}(P)|^p$$

where $\overrightarrow{\nabla}(P)$ is the gradient at the sampled point P, o_o is the original sample opacity, and o_e is the enhanced sample opacity.

Silhouette enhancement: Silhouettes are useful for orientation cues and for rendering a sketch of the feature shape. The silhouette term is strongest at areas where the view vector is orthogonal to the surface normal vector. Thus, the opacity enhancement term becomes:

$$o_e = o_o \cdot (1 - |(\overrightarrow{\nabla'}(P) \cdot \overrightarrow{V})|)^p$$

$\overrightarrow{\nabla'}(P)$ is the normalized gradient at the sampled point P and $\overrightarrow{V}$ is the normalized eye-vector.

Tone shading: Tone shading is an extension of the traditional illumination model used to convey shape by giving surfaces facing the light source "warm" colors, while other surfaces get "cooler" colors. The color modification term becomes the following:

Enhancements	Per-voxel data required
Boundary	Gradient magnitude
Silhouette	Gradient direction
Lighting	Gradient direction
Tone shading	Gradient direction
Distance color blending	Eye-space z coordinate
Distance color cue	Eye-space z coordinate
Feature halos	Halo effect intensity

Table 1. Required per-voxel information for the enhancements

$$b_{wc} = (1 + (\overrightarrow{\nabla}'(P) \cdot \overrightarrow{L}))$$
$$Color = b_{wc} \cdot Color_{warm} + (1 - b_{wc}) \cdot Color_{cool}$$

where $\overrightarrow{L}$ is light direction.

Distance color blending: This technique essentially dims volume sample colors as they recede from the viewer. At the front of the volume, voxel color remains unchanged, but as the depth value increases, the color is gradually blended with the background color. Using blue as the background color gives a result resembling aerial perspective.

$$Color = (1-d) \cdot Color_{original} + d \cdot Color_{background}$$

where d is screen-depth of the current voxel.

Distance color cues: It is common practice to convey depth information through color, so we can have the enhancement that maps screen depth (eye distance) to an arbitrary color map.

Feature halos: Feature halos are the enhancements that highlight strong features by putting a null "halo" around them, i.e. by darkening the region around the features. The next section describes the calculations required for the halos and reviews our new algorithm.

Having this set of basic enhancements, it is possible to combine them for more effective enhancement (e.g., applying boundary, sihouette, and tone shading simultaneously). The shaders for such combinations can be easily derived from the basic enhancement shaders with additional parameters allowing the interactive adjustment of the strength of each enhancement.

All of the enhancements require extra data per voxel. Table 1 summarizes the values needed for each specific enhancement. To determine the gradient, we use the gradient table that is stored as a separate texture and used inside the shader. We generate it during pre-processing, using the central difference technique [8]. There are many alternative ways for gradient estimation and a survey of these techniques can be found in [13]. Another option is calculating central differences within the fragment shader, since modern hardware places virtually no restrictions on how many texture lookups we can do in a single rendering pass. This approach requires 50% less texture memory and uses floating-point precision in the calculations. However, in order to sample volume at the neighbor voxels, seven texture lookups are required instead of two (for the pre-generated gradient option).

The eye-space z coordinate, mentioned in Table 1, is passed to the shader through the vertex parameters of the slice. The halo value is view-dependent, and cannot be precomputed. However, a portion of the halo calculation is view independent as described in the following section.

3.3. Feature halos

Feature halos make regions around strong features darker and more opaque, obscuring the background elements which would otherwise be visible. These "null" halos are effective for making important features stand out from the background. The strongest halos are created in the empty regions just outside (in the plane perpendicular to the view direction) the strong feature, so we examine the immediate neighborhood of the voxel in calculating feature halos. The algorithm suggested in [5] calculates an extra value per voxel, the halo-effect intensity H, using Equation 1:

$$H_i = (\sum_{j}^{neighbours} dw_j h_{ij}) \cdot (1 - |\overrightarrow{\nabla}(P_i)|) \qquad (1)$$

dw_{ij} are the weights of each neighbor's halo influence and are inversely proportional to the square distance between the considered voxel and the neighbor. h_{ij} is the maximum potential halo contribution of a neighbor j with respect to voxel i, given by Equation 2:

$$h_{ij} = |\overrightarrow{\nabla}(P_j) \cdot \vec{e_{ij}}|^{k_{hpe}} (1 - |\overrightarrow{\nabla}'(P_j) \cdot \overrightarrow{V})|)^{k_{hse}} \qquad (2)$$

Here, $\overrightarrow{\nabla}(P_j)$ is the neighbor's gradient, $\overrightarrow{\nabla}'(P_j)$ is the normalized neighbor's gradient, k_{hpe} controls how directly the neighbors gradient must be oriented toward the current location, and k_{hse} controls how tightly halos are kept in the plane orthogonal to the view direction $\overrightarrow{V}$. The sample is then modified according to the magnitude of the halo-effect, usually increasing the opacity and decreasing the color proportional to H.

Calculation of the halo-effect intensity can be implemented in the fragment shader; however, it would be inflexible (the neighborhood cannot be changed once the shader is written), involve excessive redundancy in calculation, and have extremely slow performance. For example, to consider the $3 \times 3 \times 3$ neighborhood of a voxel we would need 27 texture lookups.

In order to overcome this obstacle, we have developed an alternative version of the algorithm, using the paradigm of multi-pass rendering. We generate the halo influence volume in the first pass and use the resulting volume texture in the second pass. According to Equation 2, every voxel potentially affects the halo values of its neighborhood. In other words, every voxel generates a spherical halo influence area around it. To determine this halo-effect intensity for each voxel, we must add up all the neighbors' influences.

We, therefore, create a *halo-buffer*, which is essentially a 3D-texture with voxels containing the value of the sum of the neighbor voxels' halo influences (i.e. each voxel in the halo-buffer holds the value for the first term in Equation 1). We calculate this by "additive rendering to volume": we traverse the voxels, determining the amount of halo-influence in each voxel's neighborhood and generate *halo-splats*. Here, the halo-splat is a fixed size sub-volume that covers the specified neighborhood. After generating the halo-splat, it is *added* to the result volume. Figure 2 shows an example of how two splats contribute to the point, using the 2D case for simplicity.

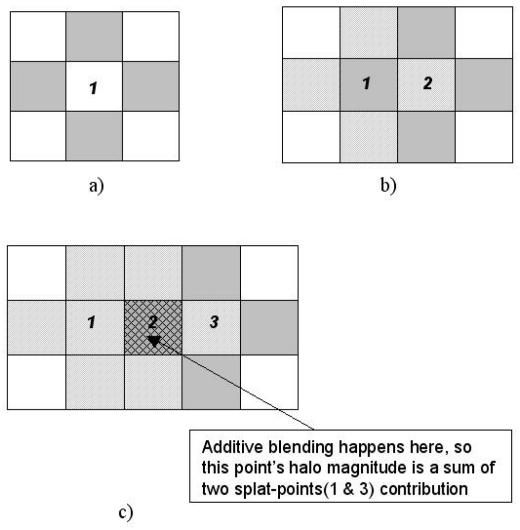

1
a)
1
2
b)
1
2
3
Additive blending happens here, so this point's halo magnitude is a sum of two splat-points(1 & 3) contribution
c)

Figure 2. Additive rendering approach for halo calculation a) one splat b) two splats c) three splats

3.4. Acceleration

To accelerate feature halo computations, we use the following three techniques. First, we incorporate a low-gradient cutoff. One of the common techniques for accelerating the volume rendering process is to eliminate non-

Dataset	0.1	0.2	0.4
Bunny	7%	4.5%	2.3%
Engine	9.5%	6.5%	1.4%
Aneurysm	1.4%	0.9%	0.5%
Head	7%	1.6%	0.1%

Table 2. Percentage of high-gradient voxels, depending on the threshold value

qualified areas of the volume from the processing. In our case, the significant halo influences are generated by the voxels with high gradient. Statistical analysis of volume data shows that the percentage of such voxels is low for volumetric datasets (see Table 2). Therefore, we can pre-process the volume and filter out voxels that will not contribute to the halo.

Second, we can calculate the halos at a lower resolution. Ideally, for every voxel in the original dataset we should have a halo-buffer value. However, if an approximate value will suffice, we can use a lower resolution halo-buffer. In this case, for each voxel in the halo-buffer, the gradient has to be interpolated using an appropriate filter. In our case, we either use a trilinear interpolation filter, or average a $2\times2\times2$ voxel block (when halo-buffer dimensions are exactly half of the original volume dimensions). Again, this filtering is done at the pre-processing stage and does not affect the interactive performance.

Finally, for the generation of the halo-buffer, we can use the graphics hardware for most of the calculations. We initialize the resulting frame buffer on the graphics card to have exactly the same number of pixels as our result 3D texture. As the algorithm uses a pixel buffer as a rendering target, which is later converted into a 3D-texture, we define a 2D to 3D mapping. One popular approach to achieve this is to layout the 3D-texture slices on a 2D-plane. An example of this mapping is in Figure 3.

We need to render halo-splats for every voxel. Since the halo-splat is essentially a small volume, we need to render a set of halo-splat slices, designated *slabs*: 2D decompositions of the splat. Thus, for a $n \times n \times n$ splat we have n slabs with size $n \times n$. Rendering of these slabs uses an approach similar to texture-based splatting [3], where splats are rendered as texturized quadrilaterals. For each slab, the rendering implements Equation 2. The result depends on the view vector, gradient vector for the current voxel. Each element of the halo-splat depends on the neighbor-vector and the weight. Thus, we need the following data:

- Per-frame information: view vector $\overrightarrow{V}$
- Per-voxel information: gradient vector $\overrightarrow{G}$
- Per-texel information (in halo buffer): vector e_{ij},

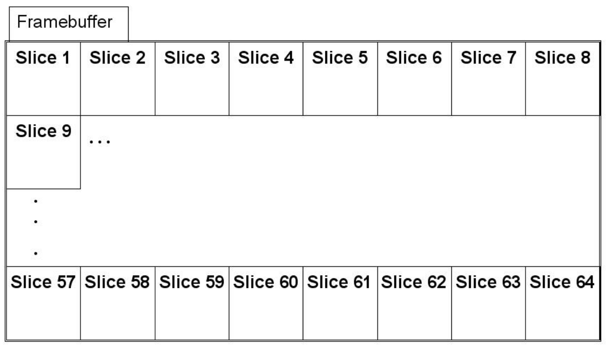


Figure 3. Mapping of 3D-texture 64x64x64 to a 512x512 framebuffer. Each slice is 64x64

weight w.

In order to provide fast calculation of the result, we create a fragment shader that takes the above parameters as input and computes the result of Equation 2.

Per-texel information is supplied by the *splat-texture*, which essentially defines the shape of the splat. The splat-texture is the same for all the voxels, and, therefore, we generate it at the initialization phase. The generation proceeds as follows: for each element inside the splat-texture, we calculate the normalized vector toward the splat center (neighbor-vector $\overrightarrow{e_{ij}}$) and the weight. The weight value is inversely proportional to the squared distance from the current voxel to the splat center, and the weights for all the voxels in the splat must sum to 1. The $\overrightarrow{e_{ij}}$ and w value are put into $RGBA$ fields of the texture, as shown in the Figure 4. The generated 3D splat-texture is then sliced up into n 2D slab-textures.

e^x_{ij}	e^y_{ij}	e^z_{ij}	w

Figure 4. Texel of the splat-texture

The next stage is the first rendering pass, which generates the halo-buffer. At this step, all the halo-splats are rendered into a pixel buffer. Since overlapping halo-splats are summed according to Equation 1, we use additive blending. For each qualified voxel in the halo-buffer, we load its gradient $\overrightarrow{G}$ as a parameter to the shader and then render each slab as a quadrilateral textured with the respective slab-texture. Thus, during each pixel's calculation of the resulting splat, the shader takes all the required data ($\overrightarrow{V}$, $\overrightarrow{G}$ as parameters, $\overrightarrow{e_{ij}}$ and w from the respective texel in the slab-texture) and calculates the halo-influence accordingly, placing the result in the $RGBA$ components of the output

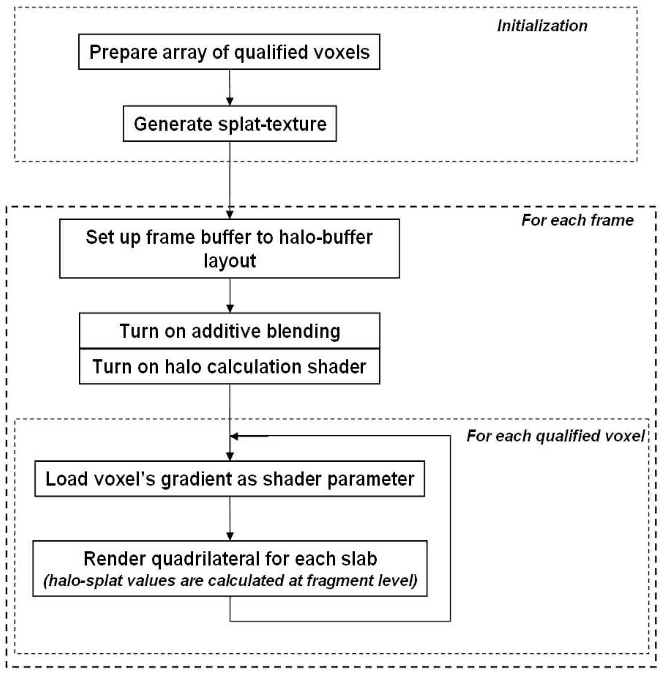

Initialization
Prepare array of qualified voxels
Generate splat-texture
For each frame
Set up frame buffer to halo-buffer layout
Turn on additive blending
Turn on halo calculation shader
For each qualified voxel
Load voxel's gradient as shader parameter
Render quadrilateral for each slab
(halo-splat values are calculated at fragment level)

Figure 5. Halo-buffer rendering flowchart

fragment. The complete process of halo-buffer generation is outlined in Figure 5.

After all the splats are rendered, we need to create a 3D-texture from the resulting pixel buffer, completing the halo-buffer creation. This is done using the standard OpenGL call `glCopyTexSubImage3D` for copying areas from the buffer into 3d-texture slices.

During the second pass, the final volume rendering shader enhancement becomes a straightforward multiplication of the halo-buffer value by $(1 - |\nabla(P_i)|)$, according to Equation 1. When the halo effect intensity H has been calculated, we apply it by increasing the opacity of the fragment by $c_1 H$, and decreasing the color by $c_2 H$, where c_1 and c_2 are constant parameters, controlling the impact of the halo on the final image.

4. Results and analysis

We have implemented the previously described volume illustration techniques and applied them to several different datasets. In our images, we have used the following datasets: *head*[1] (CT scan of a human head), an *engine*[2] (CT scan of an engine block), a *rabbit*[3] (CT scan of a hollow clay

[1]UNC Chapel Hill
[2]http://www.volvis.org
[3]National Library of Medicine

rabbit figurine), and an *aneurysm*[4] (CT scan of the arteries with a contrast agent, with aneurysm) dataset. Figure 6 shows results of applying the different enhancements described in the paper. The first row compares non-enhanced volume rendering, rendering with color transfer function and boundary enhancement, and rendering with silhouette enhancement; the second row presents our results in applying illumination, tone shading, and combined silhouette/boundary enhancement. Figure 7 shows the application of feature halos, along with boundary enhancement.

The performance (i.e., the frame rate) of the particular enhancement implementation depends on the dataset size, image resolution, fragment shader complexity, and the number of fragments to render.

Essentially, to render an image, we need to render N slices. During the shading of each slice, the current fragment program is executed for each pixel in the slice. Thus, it is reasonable to expect the rendering to be proportional to the number of slices, the screen area, and the complexity of the fragment program code.

Fragment program execution time obviously depends on the number of instructions generated by the compiler and the type of those instructions. However, since we do not have detailed information about fragment program loading and execution, we cannot predict exactly how much time a certain program will take. Generally, we can compare the performance of the program by either benchmarking or approximately comparing programs using the number of standard operations (dot-products, texture lookups) used in them.

The number of slices can serve as the main quality-vs-performance parameter. A lower number of slices increases performance; however, this may lead to *slicing artifacts*, where the volume is not sampled with sufficient frequency.

We have tested our unoptimized system with a number of volumetric datasets, resulting in very good performance. For a typical volume size of 256 x 256 x 128, we achieve frame rates of 6 to 10 frames per second for the gradient enhancement, about 4 frames per second for the tone shading, and 4 to 5 frames per second for the halo rendering and gradient enhancement combined. With optimization, we believe we should be able to achieve a 50% to 100% performance increase. Our tests were performed on the Pentium 4 1.5 Ghz PC with 1.5 Gbytes of RAM and a GeForce FX 5800 Ultra card (128 Mbytes of VRAM), using 250 slices, and a screen area of about 300×300.

5. Conclusions and Future Work

We have presented an interactive volume illustration system, which incorporates many feature enhancements, provides interactive exploration of volume datasets and features, and allows interactive adjustment of enhancement parameters. This new system is a very powerful exploration, previewing, analysis, and educational tool. Most common volume illustrative enhancements are part of the system and can be easily combined to achieve the most appropriate rendering of the volume dataset. Our new halo-buffer algorithm is a very fast alternative to traditional halo generation and this approach can be used for other local area enhancements and for volumetric shadow buffer creation.

[4]Philips Research, Hamburg, Germany

6. Acknowledgements

This material is based upon work supported by the US National Science Foundation under grants: NSF ACI-0081581, NSF ACI-0121288, NSF IIS-0098443, NSF ACI-9978032, and NSF ACI-0222675. We would also like to thank our colleagues, Wei Qiao and Kirk Riley, for valuable help with software development.

References

[1] *Cg Toolkit, a Developer's Guide to Programmable Graphics*. NVIDIA, Santa Clara, CA, 2002.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *ACM Symposium on Volume Visualization*, 1994.

[3] R. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In *IEEE Visualization '93 Proceedings*, pages 261–266. IEEE Computer Society, 1993.

[4] B. Csébfalvi, L. Mroz, H. Hauser, A. König, and E. Gröller. Fast visualization of object contours by non-photorealistic volume rendering. *Computer Graphics Forum*, 20(3):452–460, 2001.

[5] D. Ebert and P. Rheingans. Volume illustration: non-photorealistic rendering of volume models. In *Proceedings of the conference on Visualization '00*, pages 195–202. IEEE Computer Society Press, 2000.

[6] A. Hertzmann and D. Zorin. Illustrating smooth surfaces. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.

[7] V. L. Interrante. Illustrating surface shape in volume data via principal direction-driven 3D line integral convolution. *Computer Graphics*, 31(Annual Conference Series):109–116, 1997.

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

[9] A. Lu, C. J. Morris, D. S. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques. In *Proceedings of the conference on Visualization '02*, pages 211–218. IEEE Press, 2002.

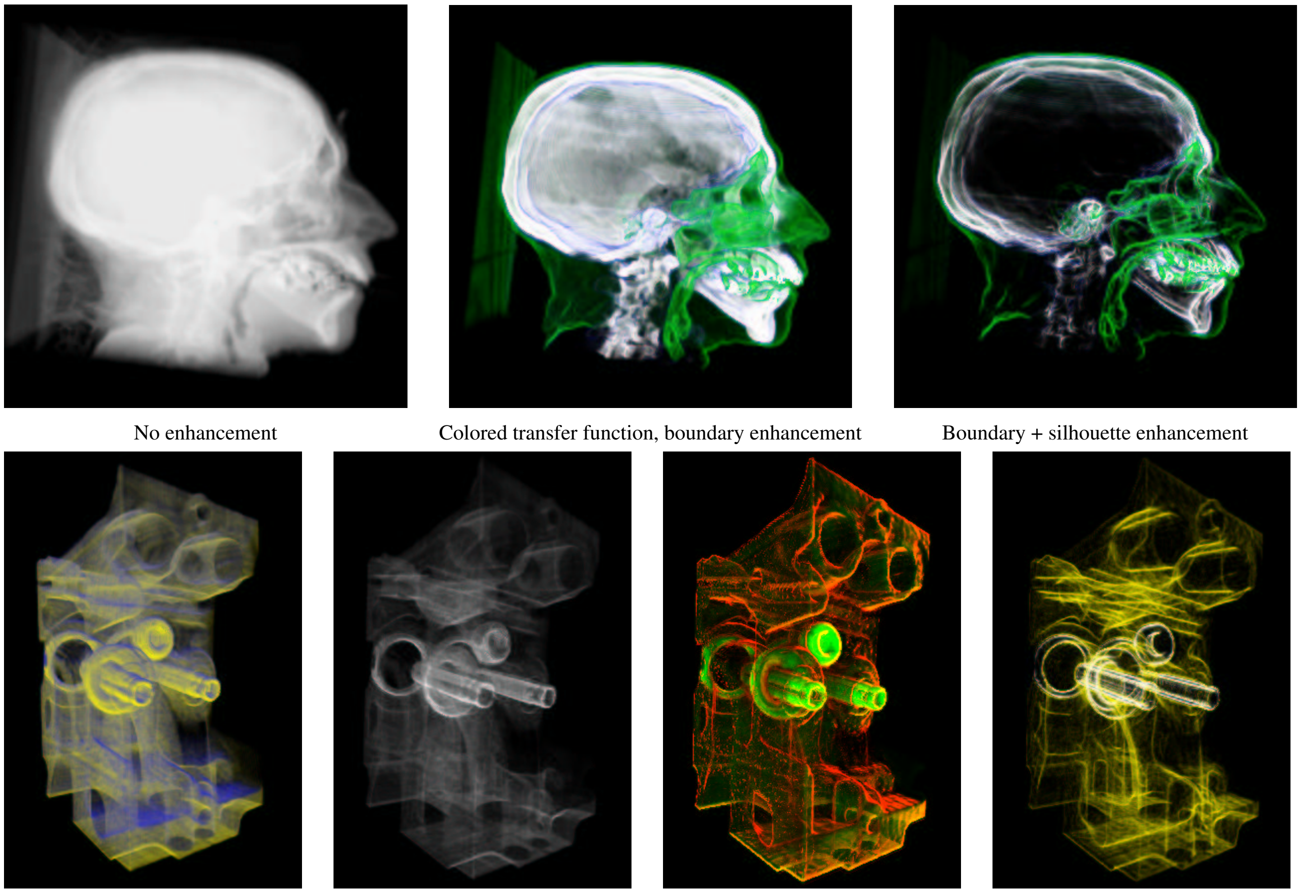
[10] E. B. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, pages 67–ff. ACM Press, 2002.

[11] E. B. Lum and K.-L. Ma. Interactivity is the key to expressive visualization. *ACM SIGGRAPH Computer Graphics*, 36(3):5–9, 2002.

[12] Z. Nagy, J. Schneider, and R. Westermann. Interactive volume illustration. In *Proceedings of Vision, Modeling and Visualization Workshop '02*, 2002.

[13] L. Neumann, B. Csébfalvi, A. König, and E. Gröller. Gradient estimation in volume data using 4d linear regression. *Computer Graphics Forum*, 19(3):351–358, Aug. 2000.

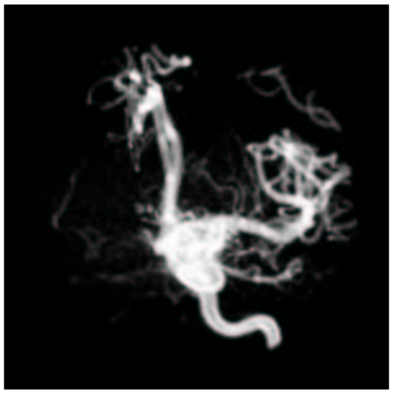
[14] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Realtime hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581. ACM Press, 2001.

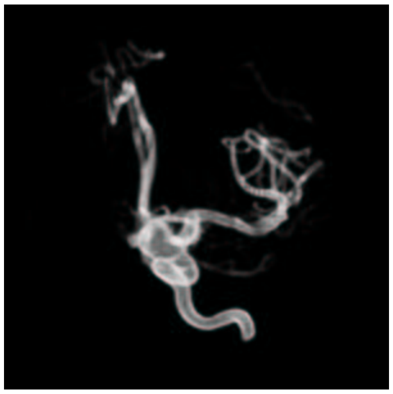


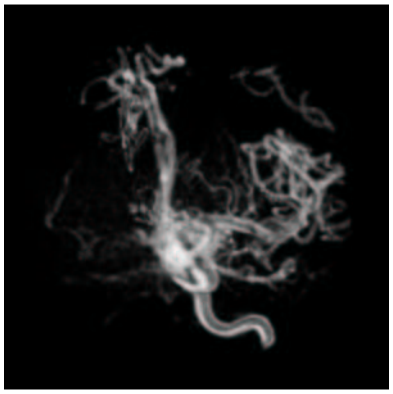
No enhancement — Colored transfer function, boundary enhancement — Boundary + silhouette enhancement

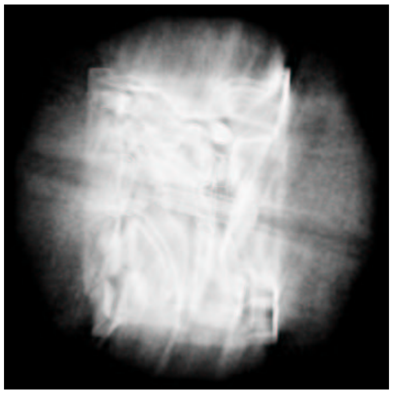
Tone shading — Phong illumination — Boundary/silhouette, colored — Silhouette

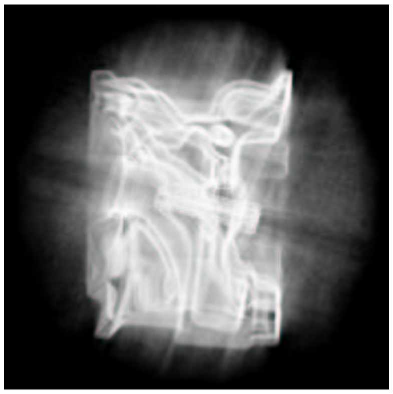
Figure 6. Example interactive volume illustration enhancements.

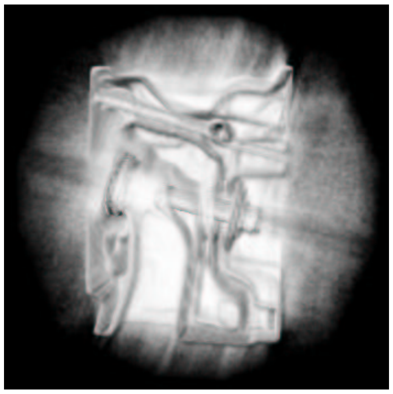

Aneurysm: no enhancements

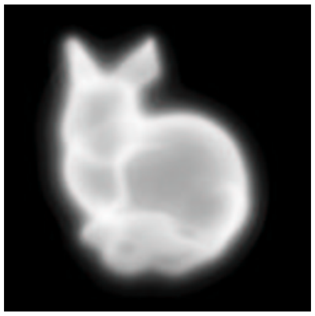

Boundary enhancement

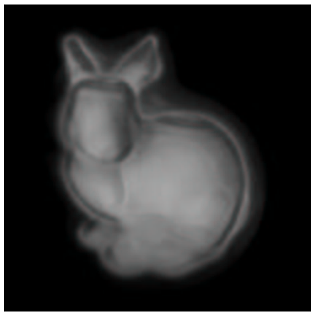

Feature halos

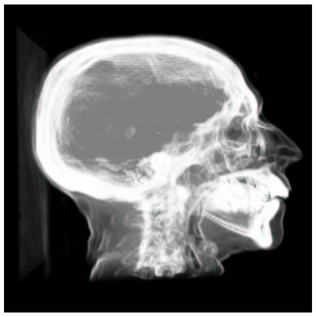

Engine: no enhancements


Boundary enhancement


Feature halos


Bunny: boundary enhancement


Feature halos


Head: highlighted silhouettes

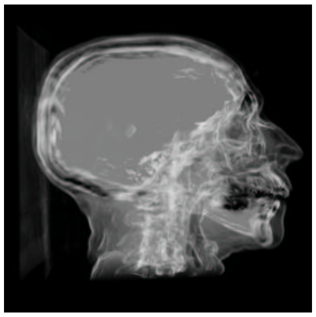

Head: silhouettes and halos

Figure 7. Feature halos effect applied to various datasets