High-Quality Silhouette Illustration for Texture-Based Volume Rendering

Zoltán Nagy, Reinhard Klein
Institut für Informatik II, University
of Bonn
Römerstraße 164
53117 Bonn
Bonn, Germany
{zoltan|rk}@cs.uni-bonn.de

ABSTRACT

We present an accurate, interactive silhouette extraction mechanism for texture-based volume rendering. Compared to previous approaches, our system guarantees silhouettes of a user controlled width without any significant preprocessing time. Our visualization pipeline consists of two steps: (a) **extraction of silhouettes** with a width of one pixel, and (b) image post-processing for **broadening of silhouettes**. Step (a) is a mixture of object- and image-based- silhouette extraction models, maximally exploiting the screen resolution. This hybrid approach is neither sensitive to accuracy in gradient representation nor to the precision of the depth-buffer, as in earlier procedures. Step (b) is accomplished via smoothing and applying a threshold to the temporary result obtained in (a). To keep the latter process efficient, we perform fast convolution using FFT. Our silhouette extraction is conceptually similar to the corresponding method for polygonal representations, checking the front- and back facing property of adjacent triangles.

Keywords

NPR, volume rendering, silhouettes, stylization, contours, FFTW.

1. INTRODUCTION

Volume rendering has become an important tool for scientific visualization in the last decade. The major focus in this area lies in the exploration of datasets as obtained from Computer Tomography (CT), Magnetic Resonance Imaging (MRI) or simulations. Iso-surface extraction and direct volume rendering (DVR) have proved themselves as interactive exploration methods for input data in texture-based volume rendering. These two methods are alike in their objectives to approximate the look of the analyzed objects as they would appear in reality: iso-surface extraction describes an opaque-like look, whereas DVR visualizes a semi-transparent appearance.

Journal of WSCG, Vol.12, No.1-3, ISSN 1213-6972
WSCG'2004, February 2-6, 2004, Plzen, Czech Republic.

Only recently, researchers have recognized the impact of combining the two areas of (i) volume rendering and (ii) non-photorealistic rendering (NPR). NPR leaves freedom to guide the attention of the observer to special features of the object, like silhouettes, creases, cusps, or material edges. For an overview of this topic and the terms mentioned above, we refer to [StSc02], [MöHa02] and [GoGo01].

This work deals with the question of how to detect and illustrate *silhouettes* in volumetric datasets efficiently and robustly. We address the problem of capturing silhouettes with a guaranteed width of one pixel and broadening of silhouettes either by a user-defined, fixed width -or adaptively, depending on the distance to the viewer.

Our paper is organized as follows. In section 2, we review related work. In section 3, we describe our method of finding silhouettes in the dataset from a particular view. Section 4 explains how the silhouettes can be broadened for advanced stylization. The remaining sections summarize results and conclude our work.

2. PREVIOUS WORK

2.1 Polygonal Models

Detecting silhouettes for polygonal representations is most simple. A well-established criterion is used, which we call the *front/backface property* of two adjacent polygons: an edge is called silhouette edge, if exactly one of two triangles sharing an edge faces the viewer [MöHa02], i.e.

$$(\boldsymbol{n_0} \cdot \boldsymbol{v} > 0) \neq (\boldsymbol{n_1} \cdot \boldsymbol{v} > 0) \quad (1),$$

must hold, where $\boldsymbol{n_0}$, $\boldsymbol{n_1}$ are the respective triangle normals and $\boldsymbol{v}$ is the viewing vector.

The union of silhouette edges forms the silhouettes. Raskar and Cohen [RaCo99] and Raskar [Ra01] used this criterion for real-time silhouette rendering. Their system is capable of illustrating silhouettes of predefined width by enlarging back-facing triangles by a depth-dependent factor. Although other techniques, like e.g. the shell (or halo) method [HaDa01] exist to render silhouettes in real-time, Raskar and Cohens method is currently regarded as being best concerning speed and robustness. As mentioned above, this criterion cannot be used directly for volumetric representations; however, we shall exploit the robustness of the *front/backface property* in our approach in a different way.

2.2 Surface Angle Silhouetting

For surface representations, where the criterion above cannot be applied, often the *right-angle criterion* is used: a point is called silhouette point, if the inequality $|\langle \boldsymbol{v},\boldsymbol{g} \rangle| < \varepsilon$ holds, where $0<\varepsilon<1$ denotes a threshold value and $\boldsymbol{v}$ and $\boldsymbol{g}$ are the normalized local viewing direction and the gradient on the surface, respectively [GoSl99]. Despite its applicability for arbitrary shape representations, a drawback is that silhouette lines are drawn with variable width, depending on the curvature of the surface [MöHa02]. Csébfalvi et al. [CsMr01] and Rheingans and Ebert [RhEb01] improved this formula by introducing a constant **k** and checking the relation $(1- |\langle \boldsymbol{v},\boldsymbol{g} \rangle|)^{\mathbf{k}} < \varepsilon$, where **k** serves the purpose of controlling the contour sharpness. By this means, above mentioned effects get lessened, but not removed, since the curvature of the surface still influences the silhouette width. Kindlmann et al. [KiWh03] try to incorporate curvature information in their model, but their method is still not robust in general, e.g. in regions where curvature is too low to be measured accurately.

2.3 Silhouettes by Image Processing

The methods mentioned so far operate on object level, i.e. the object geometry is used for silhouette detection. Discontinuities in screen-space, however, can also be used for detecting boundaries. Saito and Takahashi [SaTa90] first picked up this idea, followed by Decaudins [De96] extension towards toon rendering. Based on the simple idea that silhouettes tend to be located rather at pixels where discontinuities in the neighborhood in the Z-buffer occur, the method works fairly well, even for non-polygonal representations. Card and Mitchell [CaMi02] and Mitchell [Mi02] improved this method by taking normal discontinuities in image space into account. There are some flaws with this technique making it disadvantageous for volume rendering. First, for nearly edge-on surfaces, the z-depth comparison-filter can falsely detect silhouette edge pixels. Second, if the differences in z-depth comparison are minimal, then silhouette edges can be missed [MöHa02]; in other words, the depth-comparison is sensitive to the resolution of the depth-buffer. Deussen and Strothotte [DeSt00] use the same z-buffer trick to generate pen-and-ink trees, therefore it is related to our silhouette extraction technique. Their algorithm however, uses a fixed threshold to determine discontinuities in z-space. Our algorithm is not restricted to an arbitrarily chosen value, but uses object-precision information to adaptively locate the outlines.

2.4 Volumetric Models

Surface angle silhouetting has been approved in volume rendering in various applications. Csébfalvi et al. [CsMr01] used it for visualizing contours, Rheingans and Ebert [RhEb01] for volume illustrations, Lu et al. [LuMo02] for point-stippling techniques, Svakhine and Ebert [SvEb03] for feature halos, Nagy et al. [NaSc02] for hatching and Lum and Ma [LuMa02] in parallel applications.

A very elegant method for extracting silhouettes, tailored to volumes, was proposed by Schein and Elber [ScEl02], who used a trivariate tensor product B-spline representation of their data to obtain highly accurate boundary renderings. Their method however, demands tremendous amounts of memory and disk space, with preprocessing times of more than 20 minutes and about 10 seconds for a particular view on a 800 MHz Pentium III for a dataset with about 315.000 voxels. Our approach, in contrast, requires no significant preprocessing, and allows for interactivity.

3. ALGORITHM OUTLINE

Our algorithm takes the regular volumetric dataset as input, without any additional information, like e.g. gradients. The rendering of the dataset is accomplished by using 3D texturing under the constraint of slicing the polygons in front-to-back fashion using iso-surface extraction.

Figure 1 summarizes the rendering process. In the first stage the program renders the single slices,

detects the contours and propagates them through the slices in order to capture silhouettes. This is explained in the following subsections. Afterwards, the content of framebuffer is read back to main memory to broaden silhouettes. This is an optional stage explained in section 4. Finally, the result is output to the framebuffer.

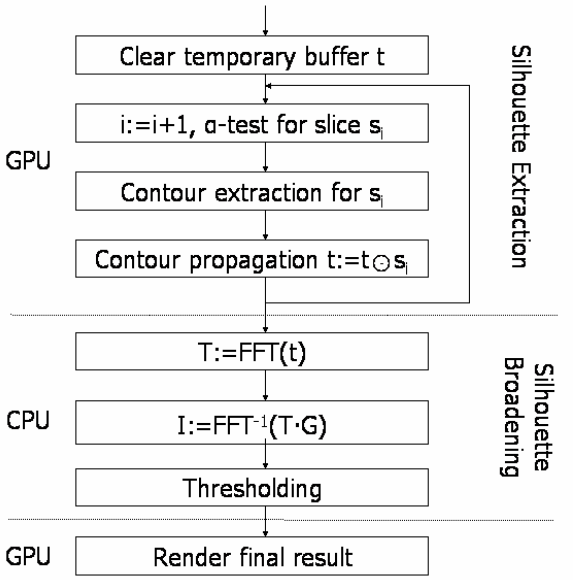

Figure 1: Survey of the rendering pipeline. **G** denotes the Fourier-transformed Gaussian kernel.

In the following subsection, we describe the idea of the first stage of our algorithm first for the 2D case, afterwards we elevate the method to 3D.

3.1 Basic Idea

The main trick of our silhouette detection mechanism is depicted in figure 2. First, we fix two terms. We define a pixel to be a *contour pixel*, if the fragment survives the α-test during rasterization, but not all pixels in the 8-neighbourhood. A contour pixel is called *propagated*, if a contour pixel was already detected on the previous slice at the same screen position.

Suppose we have a single *visible* contour pixel $\mathbf{c_k}$ on slice $\mathbf{s_{i+1}}$ detected at a particular screen position (fig. 2, top left). To decide, whether $\mathbf{c_k}$ is a silhouette pixel, we check whether a pixel is rendered for slice $\mathbf{s_{i+2}}$ at the same screen position. If this is not the case, we can assume to have a silhouette pixel detected. If multiple contour pixels are found on successive slices at the same screen position (i.e. we have propagated contour pixels), the local viewer direction is orthogonal to the iso-surface normal and we come to a decision by means of the contour pixel found farthest from the viewer at the same screen position (fig 2, top right and bottom left). If a survived fragment is found on the next slice at the corresponding position, we do not have a silhouette, otherwise we do.

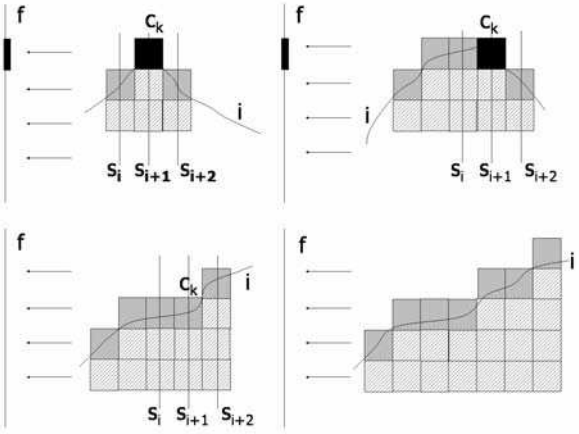

Figure 2: Examples for silhouette pixel determination. $\mathbf{s_i}$, $\mathbf{s_{i+1}}$ and $\mathbf{s_{i+2}}$ are screen-parallel slices after surviving the α-test. **f**, $\mathbf{c_k}$ and **i** denote the framebuffer, the regarded contour pixel and the iso-surface, respectively. We color code passed fragments bright grey, contour pixels grey and silhouette pixels black. Top left: $\mathbf{c_k}$ in $\mathbf{s_{i+1}}$ is detected as silhouette pixel, since it is visible and the successor fragment in $\mathbf{s_{i+2}}$ does not pass the α-test. Top right: similar situation, where $\mathbf{c_k}$ is a propagated (see text for definition) contour pixel. Bottom left: $\mathbf{c_k}$ is not recognized as silhouette pixel, since the subsequent fragment in $\mathbf{s_{i+2}}$ passes the α-test. Bottom right: importance for the decision of the definition on the bottom left: if we would define a contour pixel to be a silhouette pixel only because it is propagated, we would get multiple silhouettes on the boundary of highly curved surfaces (here: 2nd and 3rd row).

Premature classification of two successive contour pixels at the same screen position as silhouette pixel would lead to multiple silhouettes near to boundaries of curved objects, which we want to prevent (fig. 2, bottom right). This special case is not properly caught by conventional methods, like by the z-buffer trick (sec. 2.3) or the right-angle criterion (see figs. 7 left and 8 and sec. 6). Figure 3 shows a simple example for silhouette tracking in 3D. It remains now to clarify the tracking of contour pixels through the single slices.

3.2 Contour Propagation

The algorithm itself works like a standard front-to-back iso-surface extraction pipeline, with extended operations applied on a single slice. Since these rules require an access to temporarily obtained results, we keep three textures in the texture units (TU) of the graphics board containing copies of the framebuffer (see table 1).

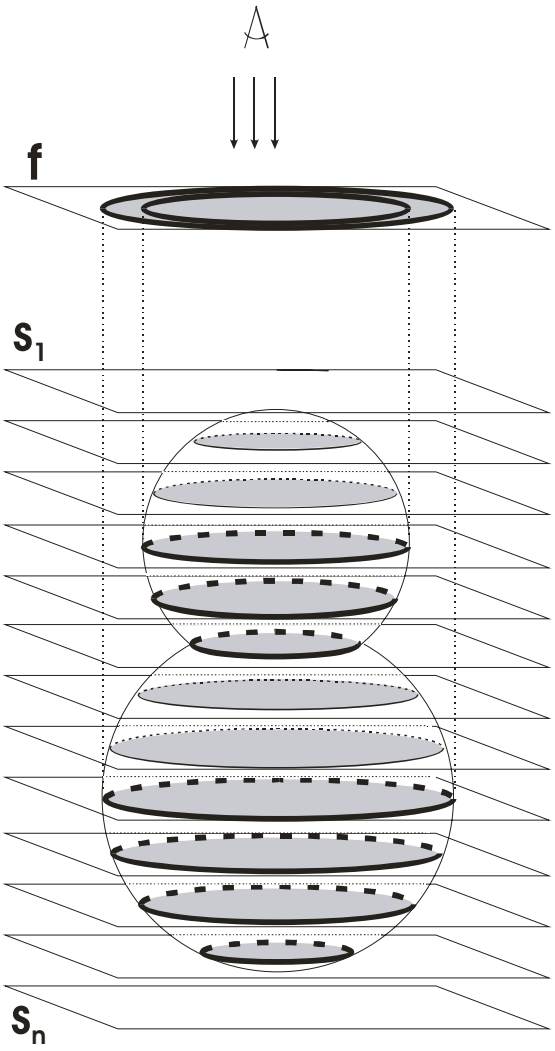

Figure 3: Silhouette determination in 3D. An object (here: two melted spheres) is rendered in a front-to-back (here: from top-to-bottom) fashion. Fragments surviving the α-test are opaque and shown in gray. When rendering the active slice, we assure not to alter passed pixels in the framebuffer **f**. If a contour pixel in slice $\mathbf{s_i}$ corresponds to an empty pixel in the subsequent slice $\mathbf{s_{i+1}}$ (at the same window position), then it is considered a potential silhouette pixel (bold outline). Due to this construction, only two silhouettes appear in the framebuffer after rendering all slices in the shown example.

TU	Content	Dim.
0	Volume Data	3D
1	Footprint	2D
2	Contour	2D
3	Intermediate Result	2D

Table1: Texture setup.

Initially, we clear texture units 1-3 with the background color. We thereafter perform the following steps, each associated with its own fragment program, in a front-to-back manner (during rendering we have *depth-testing and depth-writing disabled*):

1. Render the active volume slice with the *α-test enabled*. Store the content of the framebuffer in TU 1 and call it *footprint*. This way, we naturally obtain two classes of pixels, defined here as *empty* ( ) and *filled* ( ).

2. Render a screen-filling quad, textured with the *footprint* in TU 1. A *filled* pixel is altered here to a *contour pixel* ( ), if *not all* pixels in the 8-neighbourhood are *filled*. We store the result in TU 2 and call it *contour*.

3. Finally, we render a screen-filling quad, textured with the *contour* in TU 2 and the *intermediate result* in TU 3. The decision table shown in table 2, with ■ denoting a *silhouette pixel*, tells us how to combine two pixels at the same texture (here: (yet) screen) coordinate from TU 2 and TU 3 to a new one, using the operator ⊙ (see fig. 4 for an example).

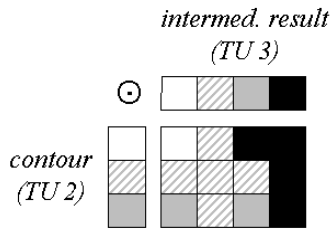

Table 2: Decision table defining how single color values in TU 2 and TU 3 are combined to a new one, stored as a new intermediate result in TU 3. The color coding used here is defined in the text.

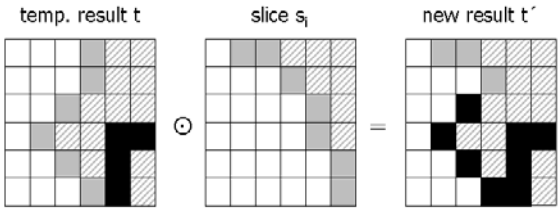

Figure 4: Example for tracking of silhouettes.

The idea behind the decision table (tab. 2) is the following:

- 1st column: Since *empty pixels* in $\mathbf{s_i}$ are transparent, they are always overdrawn by pixels in $\mathbf{s_{i+1}}$.
- 2nd column: Since *filled pixels* in $\mathbf{s_i}$ are opaque, they are never overdrawn by pixels in $\mathbf{s_{i+1}}$.

- 3rd column: Here we actually detect silhouette pixels the first time, if existent. 1st row: detection, as explained on top of fig. 2. 2nd row: no silhouette pixel, as explained on bottom of fig. 2. 3rd row: contour pixel propagation.
- 4th column: *Silhouette pixels* determined on $\mathbf{s_i}$ are unconditionally propagated to all subsequent slices.

The new result after step 3 is stored in TU 3 as the new intermediate result t'. We repeat steps 1-3 until all slices are processed. We may render an additional empty slice, if the iso-surface of the object cuts the parametric domain of the volume to ensure that contours on the last slice are discovered as silhouettes, if necessary. Due to the decision table (see first *row*), the final result in TU 3 contains only *empty*, *filled* and *silhouette pixels*, which are finally swapped into the front buffer.

Using this procedure, artifacts can occur if the interslice distance is chosen too high, emanating from places, where the slope of the surface over the image plane is too high. These artifacts can be removed by increasing the slicing density completely or adaptively; latter issue is not integrated in our framework yet.

We might also shorten the rendering cycle by using fewer steps than the described three. This would result in much longer fragment programs, which we wanted to circumvent in our current implementation for reasons of clarity and implementation ease. Even more, fewer fragment programs would not guarantee better performance, since a workaround would lead to a massive increase of fragment instructions executed per pixel.

So far we have discussed how we can precisely locate and extract silhouettes with a thickness of exactly one pixel. With a slight modification in the fragment program and by extending the rendering pipeline on the CPU, this method can be expanded to process more sophisticated silhouettes.

4. SILHOUETTE BROADENING

In the previous section, we have generated silhouettes with a guaranteed width of exactly one pixel. For many types of illustrations, especially in stylization, it is required to have silhouettes with a thickness either predefined, or depending on the distance to the viewer, to create exact controllable depth-cues or atmospheric effects [StSc02]. In this section, we insert a post-processing filtering step into the rendering pipeline, which accomplishes this task.

After rendering steps 1-3 in the previous section, the content of the framebuffer is low-pass filtered. This leads to a diffusion of silhouette lines by a clearly defined amount, in direction of the image-space gradients of the silhouettes. This can be done simply by applying a Gaussian filter on the source image using convolution:

$$f'(x,y) := f(x,y) \otimes g(x,y) \quad (2),$$

where *f'* is the new smoothed version of *f* using the two-dimensional Gaussian kernel

$$g(x,y) = \frac{1}{2\pi} e^{-\frac{1}{2}(x^2+y^2)} \quad (3).$$

The resulting image *f'* is not a bi-level image any more. By carefully converting the grayscale image *f'* back into a bi-level one, we can exploit the continuous run of the co-domain in *f'* to query the width of the silhouette at a particular pixel position. Since (3) is radial symmetric, we can rewrite it in polar coordinate representation as

$$g(r) = \frac{1}{2\pi} e^{-\frac{1}{2}(r^2)} \quad (4).$$

Furthermore, *g(r)* is monotonically decreasing (and thus invertible with inverse function g^{-1}) in the respective intervals $(0,\pm\infty)$, so we can retrieve the distance *r* of a pixel with gray tone *h* from the center of a silhouette by testing the relation

$$r = g^{-1}(h) < r_{th} \quad (5),$$

where r_{th} is half of the width of the silhouette. This solution is appealing for two reasons.

First, due to the convolution, we can accomplish the filtering process fast, and independently of the size of the convolution kernel using the well-known identity

$$f \otimes g = \mathscr{F}^{-1}(\mathscr{F}(f) \bullet \mathscr{F}(g)) \quad (6),$$

where $\mathscr{F}$ and $\mathscr{F}^{-1}$ denote the Fourier transform and its inverse, respectively. In this way, we can low-pass-filter the image with a performance independent of the size of the (discretized) Gaussian kernel. This proceeding clearly outperforms the `glConvolutionFilter2D` function of OpenGL, which permits interactivity for yet small kernel sizes. For small kernel sizes (like e.g. 8x8), however, hardware-based filters -as described e.g. in [ViKa03] or [HaBe03] - might perform better.

Second, we can control the thickness of the silhouettes adaptively, depending on the distance of the fragment to the viewer, producing the desired atmospheric effects. Thus, we abandon the idea that *f(x,y)* is bi-level and code the aforementioned distance of a fragment to the viewer in grayscale. The convolution process therefore induces a faster

decrease of intensity in direction of the screen-space gradient in *f'(x,y)*, where the original silhouette pixel color in *f(x,y)* resembles more the background color. Applying a constant threshold over the whole image *f'(x,y)* gives the desired atmospheric effect.

5. IMPLEMENTATION DETAILS

Our implementation is based on the OpenGL, GLUT and FFTW [FFTW98] libraries in C/C++.

Since rasterization remains the main bottleneck in our application (see also table 3), we do not lay special emphasis on experimenting with a hardware implementation of the FFT, as e.g. done in [MoAn03], but use FFTW instead, which is a convenient and sophisticated substitute. By comparing our approach with [MoAn03], we found that a pure hardware implementation is not necessarily a gain, especially if a powerful CPU used in combination with large screen sizes. The performance measurements below show further, that post-processing plays only a negligible role in rendering speed. Furthermore, we can smoothly integrate zero-padding [PrTe92] in the filtering process without special implementation efforts. Zero-padding is required to prevent wrap-around and thus periodic filtering of the image signal using Fourier-based, fast convolution. This is especially important when the rendered result of the object with its broadened silhouettes is not fully contained in the window.

6. RESULTS AND DISCUSSION

We have tested typical datasets (most of them available from [VoRe]) on our target platform, a Windows XP PC with a 3 GHz Pentium P4, 1 GByte RAM and an ATI Fire GL X1 graphics card. Table 3 shows performance evaluations of our method, comparing traditional right-angle criterion (RA) with our method for one-pixel width (OPS) and with post-processed, broadened silhouettes (BS). We can observe a performance loss of factor >6 on average, compared to the traditional method (last column).

We recall that our goal is to extract and visualize the *exact position* of the silhouettes on a given object and viewpoint. We do not intend to include additional clues into the rendition, like e.g. half-toned shading, etc. The bonsai tree in the left of figure 7 might convey the curvature of the local surface in a superior manner, but it does not have an exact controlled width.

Figures 5 and 6 show the Engine and the NegHip datasets, respectively, rendered (i) with the conventional right-angle criterion, (ii) with our method and (iii) with additional silhouette enhancement. Figure 7 shows the Bonsai and Skull datasets, rendered using the right-angle criterion and our method, respectively.

Dataset	Size	Win. Size	RA (fps)	OPS (fps)	BS (fps)	RA/ OPS
Bonsai	256^3	256^2	39.37	5.82	3.24	6.76
		512^2	10.00	1.60	1.04	6.25
Engine	$256^2 \cdot 128$	256^2	54.13	8.47	4.27	6.39
		512^2	13.78	2.30	1.27	5.99
Hydrog. Atom	128^3	256^2	39.37	5.98	3.51	6.58
		512^2	9.84	1.60	1.02	6.15
NegHip	64^3	256^2	19.95	3.05	2.26	6.54
		512^2	9.85	1.58	1.07	6.23
Skull	256^3	256^2	39.41	5.82	3.61	6.77
		512^2	9.85	1.58	1.06	6.23
Teddy	$128^2 \cdot 64$	256^2	44.0	5.41	3.37	8.13
		512^2	11.12	1.47	0.94	7.56

Table 3: Performance measurements for various datasets.

The results for the conventional method show also that undesired effects (non-silhouette areas) appear in the images. These artifacts appear as we are not able to determine the exact position of silhouettes due to the limitations of discrete gradient representation. This is especially perceivable in figure 8, where the gradient-method fails at near-silhouette positions on the nose of the teddy. The arrows on the right of figure 8 indicate the viewing direction and show that silhouettes must not be drawn around the nose of the teddy. The example also demonstrates the resistance of our method against inaccuracy due to coarse discretization of the dataset. The examples confirm the robustness of the special case explained in fig. 2 bottom right.

Based on the results we found our method to be more appealing as the silhouettes appear exactly at the positions we expect them to be. Furthermore, since the silhouettes initially have a width of one pixel, with the extension presented in section 4 the user can exactly control the thickness. In figure 9 we show how the widths of the silhouettes of the Hydrogen Atom dataset alter as the viewer moves closer to the object. Since broadening of silhouettes works in image-space, silhouettes can be washed out, as their density increases, e.g. when the distance of the object to the viewer becomes high.

7. CONCLUSIONS

In this paper, we have introduced a new methodology of silhouette extraction for texture-based volume rendering. It serves the purpose of visualizing silhouettes with an accurate width of one pixel. In a subsequent step, we can optionally broaden

silhouettes, either by a fixed pixel width, or depending on screen-space depth using image-processing. Our algorithm is in particular insensitive to coarse discretization in the dataset.

Silhouette detection is solved using a new paradigm, which combines accuracy at object- and screen-space resolution. We can perform silhouette enhancement in a subsequent image processing step and illustrate even exaggerated thick silhouettes –independently of their width at constant, interactive framerates.

The proposed method helps to illustrate iso-surfaces of scientific datasets in a fast fashion, allowing high degree of interactivity in rendering and modification of iso-values.

8. REFERENCES

[CaMi02] Card, D. and Mitchell, J. *Non-Photorealistic Rendering with Pixel and Vertex Shaders*. In Engel, Wolfgang, ed. ShaderX, Wordware, 2002.

[CsMr01] Csébfalvi, B. and Mroz, L. and Hauser, H. and König, A. and Gröller, E. *Fast visualization of object contours by non-photorealistic volume rendering.* Computer Graphics Forum 20(3), pp. 452-460, 2001.

[De96] Decaudin, P. *Cartoon-Looking Rendering of 3D-Scenes*. TR INRIA 2919 Université de Technologie de Compiègne, France, 1996.

[DeSt00] Deussen, O. and Strothotte, T. *Computer-Generated Pen-and-Ink Illustration of Trees*. Computer Graphics (SIGGRAPH '00 Proceedings), pp. 13-18, 2000.

[FFTW98] Frigo, M. and Johnson, S.G. *FFTW: An Adaptive Software Architecture for FFT.* ICASSP conference proceedings (vol. 3, pp. 1381-1384), 1998. http://www.fftw.org

[GoGo01] Gooch, B. and Gooch, A. *Non-Photorealistic Rendering*. A K Peters, 2001.

[GoSl99] Gooch, B. and Sloan, P.-P. and Gooch, A. and Shirley, P. and Riesenfeld, R. *Interactive Technical Illustration.* Symposium on Interactive 3D Graphics. pp. 31-38, 1999

[HaBe03] Hadwiger, M. and Berger, C. and Hauser, H. *High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware*. IEEE Visualization 2003.

[HaDa01] Hart, E. and Gosselin, D. and Isidoro, J. *Vertex Shading with Direct3D and OpenGL*. Game Developers Conference. 2001.

[KiWh03] Kindlmann, G. and Whitaker, R. and Tasdizen, T. and Möller, T. *Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications.* IEEE Visualization 2003.

[LuMo02] Lu, A. and Morris, J. and Ebert, D. and Rheingans, P. and Hansen, C. *Non-photorealistic rendering using stippling techniques.* IEEE Visualization, pp. 211-217, 2002.

[LuMa02] Lum, E. and Ma, K.-L. *Hardware-accelerated parallel nonphotorealistic volume rendering.* International Symposium on Nonphotorealistic Rendering and Animation (NPAR 02'), 2002.

[MoAn03] Moreland, K. and Angel, E. *The FFT on a GPU*. Graphics Hardware 2003.

[Mi02] Mitchell, J. *Image Processing with Pixel Shaders in Direct3D*. In Engel, Wolfgang, ed. ShaderX, Wordware, 2002.

[MöHa02] Akenine-Möller, T. and Haines, E. *Real-Time Rendering, 2nd Ed*. A K Peters, pp. 289-312, 2002.

[NaSc02] Nagy, Z. and Schneider, J. and Westermann, R. *Interactive Volume Illustration*. Vision, Modeling and Visualization 2003, pp. 497-504, 2002.

[PrTe92] Press, W.H. and Teukolsky, S.A. and Vetterling, W.T. and Flannery, B.P. *Numerical Recipes in C, 2nd ed*. pp. 496-608, 1992.

[Ra01] Raskar, R. *Hardware Support for Non-Photorealistic Rendering*. ACM SIGGRAPH/ Eurographics Workshop on Graphics Hardware, pp. 41-46, 2001.

[RaCo99] Raskar, R. and Cohen, M. *Image Precision Silhouette Edges*. Symposium in Interactive 3D Graphics, pp. 135-140, 1999.

[RhEb01] Rheingans, P. and Ebert, D. *Volume illustration: Nonphotorealistic rendering of volume models.* IEEE Transactions on Visualization and Computer Graphics, 7(2), pp. 109-119, 2001.

[SaTa90] Saito, T. and Takahashi, T. *Comprehensible Rendering of 3-D Shapes.* Computer Graphics (SIGGRAPH '90 Proceedings), pp. 197-206, 1990.

[ScEl03] Schein, S. and Elber, G. *Extraction of Silhouette Curves from Volumetric Data Sets.* The 4th Israel-Korea Bi-National Conference on Geometric Modeling and Computer Graphics, pp. 100-104, 2003.

[SvEb03] Svakhine, N.A. and Ebert, D.S. *Interactive Volume Illustration and Feature Halos*. Pacific Graphics 2003.

[StSc02] Strothotte, T. and Schlechtweg, S. *Non-Photorealistic Computer Graphics*: *Modeling, Rendering and Animation.* Morgan Kaufman.

[ViKa03] Viola, I. and Kanitsar, A. and Gröller, E. *Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages.* IEEE Visualization 2003.

[VoRe] `www.volren.org`

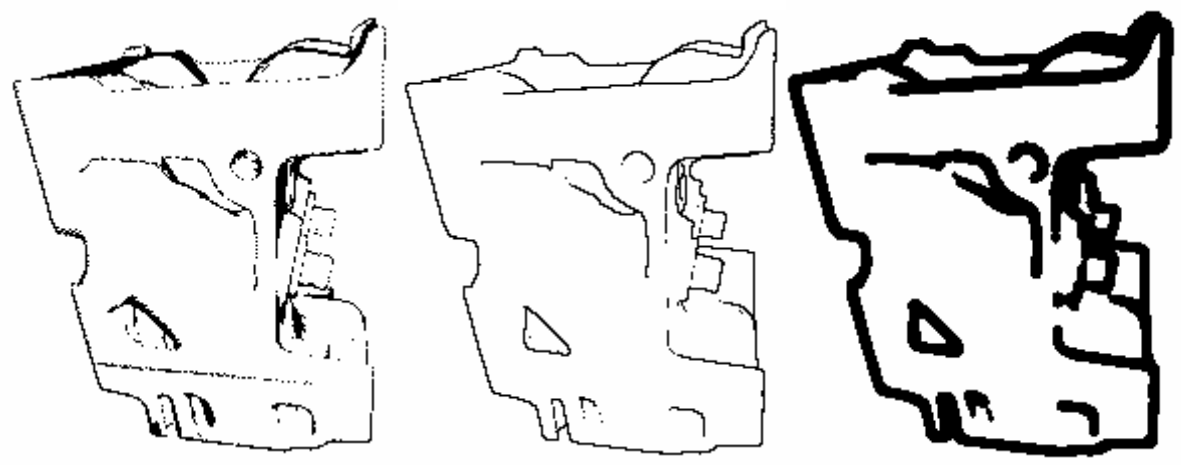

Figure 5: Engine dataset. Left: Right-angle method. Middle: our method. Right: our method with silhouette enhancement

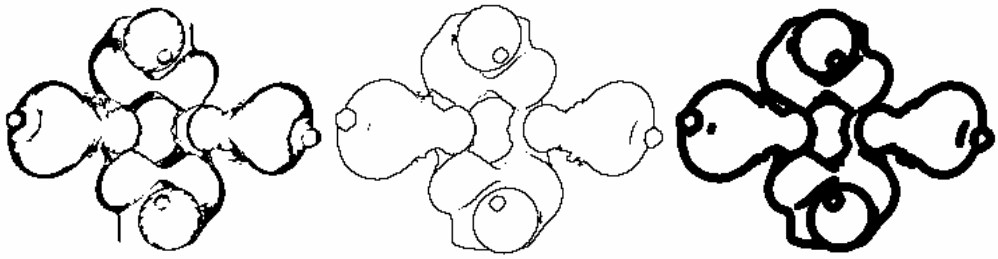

Figure 6:NegHip dataset. Left: Right-angle method. Middle: our method. Right: our method with silhouette enhancement

Figure 7: From left to right: bonsai tree with right-angle- and our method, same comparison for the skull dataset. We recall that the thick silhouette on the lower portion on the bonsai tree on the left is an unintended feature here (see text above).

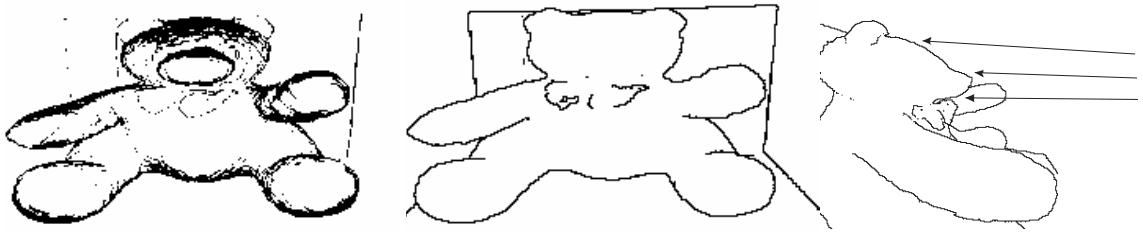

Figure 8: From left to right: teddy with right-angle- and our method; side view illustrating that silhouettes around the nose must not be drawn when the teddy is viewed from front.

Figure 9: Effect of depth-cueing on close-up, exemplified on the Hydrogen Atom dataset. Note the silhouettes becoming thicker as the object gets magnified.