Illustrative Display of Hidden Iso-Surface Structures

Conference Paper · November 2005

DOI: 10.1109/VISUAL.2005.1532855 · Source: IEEE Xplore

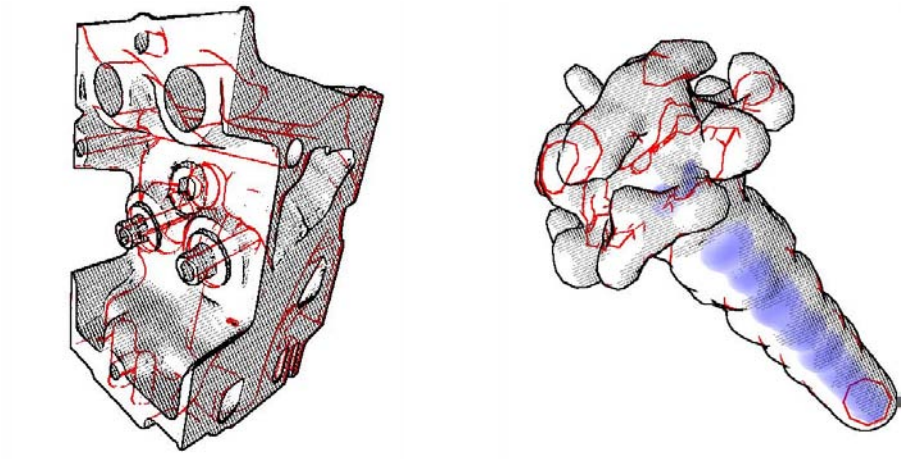
Please see supplementary material on conference DVD.

Illustrative Display of Hidden Iso-Surface Structures

Jan Fischer *
Visual Computing for Medicine
WSI/GRIS
University of Tübingen

Dirk Bartz
Visual Computing for Medicine
WSI/GRIS
University of Tübingen

Wolfgang Straßer
WSI/GRIS
University of Tübingen



(a) *Engine* dataset (b) *Fuel* dataset

Figure 1: Two images generated with our illustrative rendering algorithm. The shape of the front layer of the geometry is represented by black silhouettes and hatching. The second layer of the iso-surface is rendered as red silhouette lines. In Figure 1(b), a portion of the volume containing very large voxel values is shown as blue "secondary" geometry. The images were rendered in real-time.

ABSTRACT

Indirect volume rendering is a widespread method for the display of volume datasets. It is based on the extraction of polygonal iso-surfaces from volumetric data, which are then rendered using conventional rasterization methods. Whereas this rendering approach is fast and relatively easy to implement, it cannot easily provide an understandable display of structures occluded by the directly visible iso-surface. Simple approaches like alpha-blending for transparency when drawing the iso-surface often generate a visually complex output, which is difficult to interpret. Moreover, such methods can significantly increase the computational complexity of the rendering process.

In this paper, we therefore propose a new approach for the illustrative indirect rendering of volume data in real-time. This algorithm emphasizes the silhouette of objects represented by the iso-surface. Additionally, shading intensities on objects are reproduced with a monochrome hatching technique. Using a specially designed two-pass rendering process, structures behind the front layer of the iso-surface are automatically extracted with a depth peeling method. The shapes of these hidden structures are also displayed as silhouette outlines. As an additional option, the geometry of explicitly specified inner objects can be displayed with constant translucency. Although these inner objects always remain visible, a specific shading and depth attenuation method is used to convey the depth relationships.

We describe the implementation of the algorithm, which exploits the programmability of state-of-the-art graphics processing units (GPUs). The algorithm described in this paper does not require any preprocessing of the input data or a manual defitinion of inner structures. Since the presented method works on iso-surfaces, which are stored as polygonal datasets, it can also be applied to other types of polygonal models.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms; I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering

Keywords: illustrative rendering, non-photorealistic rendering, transparency, indirect volume rendering, hatching, shading language

1 INTRODUCTION

Volumetric datasets have become a widely used format for storing three-dimensional information in application areas like medicine and scientific visualization. The efficient display of such datasets has been a field of active research for many years. One very common method for rendering volume data is the use of polygonal iso-surfaces, which represent boundary surfaces defined by a constant intensity value within the volume. The Marching Cubes algorithm proposed by Lorensen and Cline is a widespread method for the extraction of such iso-surfaces from a volume dataset [13]. Standard polygon rasterization techniques are then used for displaying the iso-surfaces. This entire process is called indirect volume rendering.

*e-mail: fischer@gris.uni-tuebingen.de

IEEE Visualization 2005
October 23-28, Minneapolis, MN, USA
0-7803-9462-3/05/$20.00 ©2005 IEEE.

In conventional indirect volume rendering, only the front layer of the iso-surface geometry is visible to the user. Parts of the model which are at a greater depth in the eye coordinate system are suppressed by hidden-surface algorithms like the Z-Buffer. However, the shape of inner structures in a dataset is of great importance for many applications. For instance, inner organ structures or back walls of cavities often are of interest for a physician inspecting an anatomical dataset. In the case of a volume dataset containing a scanned engine block (see Fig. 1(a)), it might be useful to view hidden mechanical parts or structural weaknesses in non-destructive material testing.

The straightforward solution for displaying hidden structures of an iso-surface is to render the entire geometry with transparent polygons. Parts of the iso-surface geometry which cover the same screen space are combined using standard (alpha-)blending mechanisms. The drawback of this approach is that the entire polygonal model becomes visible. This can produce a visually complex and difficult to interpret graphical output, if too many structures are shown behind each other. Moreover, it is necessary to sort the graphical primitives according to their screen space depth, so that the blending computations yield correct results. This usually leads to a significantly increased computational complexity of the rendering process.

Direct volume rendering, which is not based on precisely defined iso-surfaces, also makes the display of interior structures of an observed dataset possible. However, the necessary definition of a useful transfer function is not trivial [20]. The images generated by direct volume rendering often show many parts of the volume dataset simultaneously, which can make viewing the structures of interest difficult for the user.

In this paper, we therefore propose a novel way of displaying hidden structures of an iso-surface. Our new algorithm utilizes illustrative rendering methods, which have become an important research area during the last years (see [28]). We have designed the new method to achieve the following advantages:

- By using illustrative visualization methods, an easily understandable graphical output is generated. Our algorithm uses silhouette outlines and monochrome hatching for conveying the shape of objects.
- The inner structures of the iso-surface are automatically extracted during the rendering process. In our approach, the first occluded layer of the iso-surface behind the front layer is displayed as hidden geometry in a distinctive silhouette-based style. No preprocessing or manual definition of objects is required.
- As an optional addition, the geometry of special inner structures of high interest can be manually specified by the user. This "secondary geometry" always remains visible and is displayed in a special solid style with depth attenuation.
- The algorithm is capable of generating real-time frame rates for most datasets and typical image resolutions.

The design of our algorithm exploits the programmability of modern graphics processing units. A specialized rendering pipeline has been developed, which integrates application-specific object space and image space processing stages. Two geometry rendering passes generate the data for the first and second layer of the iso-surface. In an optional third pass, the information required for displaying the "secondary geometry" is gathered. Finally, an image-space processing step combines all the data collected in the previous steps for achieving the desired graphical output. Our implementation uses the OpenGL Shading Language (GLSL) [24] and can deliver real-time frame rates in most cases.

2 Related Work

As mentioned above, non-photorealistic rendering has been in the focus of active research for several years. Overviews of non-photorealistic and artistic techniques have been given by Gooch and Gooch [9] and Strothotte and Schlechtweg [28]. The motivation for using non-photorealistic and illustrative rendering in scientific visualization is to emphasize visual information over physical realism. This principle has been described as "functional realism" by Ferwerda [5].

A very important basic principle employed by many non-photorealistic techniques is the generation and processing of intermediate image-space buffers containing geometric properties of the observed scene. Depth values and transformed normal vectors, which are computed for each image pixel, are frequently used geometric properties. This basic principle was introduced as *G-buffers* by Saito and Takahashi [25]. The algorithm presented in our paper is also partly based on such intermediate image-space data. Decaudin has described a method for the cartoon-like rendering of 3D objects using depth and normals buffers [2]. Mitchell et al. have presented a GPU-based technique for the extraction of object outlines based on image-space information [16]. Nienhaus and Doellner use G-buffers to create different non-photorealistic styles [17]. A framework for mapping G-buffer concepts to modern graphics processing units was described by Eißele et al. [4].

Several researchers have proposed to use colors for conveying the shape of objects or their material properties. Examples include the work of Gooch et al. on non-photorealistic lighting for automatic illustration [8], as well as Lum and Ma's watercolor inspired method for rendering surfaces [15]. Our new approach uses discrete, user-defined colors in order to distinguish the front layer of the iso-surface from the second layer and for highlighting the "secondary geometry".

Monochrome hatching and halftoning are often used in non-photorealistic rendering. These methods map a continuous range of intensities to monochrome representations based on regular patterns or artistic drawing styles. Interrante et al. have discussed the use of a texture containing discrete strokes for conveying the shape of an iso-surface [11, 12]. Hertzmann and Zorin have described methods for the line-art rendering of smooth surfaces [10]. Praun et al. have proposed *tonal art maps* as a primitive for generating an artistic monochrome hatching for 3D objects [21]. An automatic generation of tonal art maps based on arbitrary input textures has been developed by Fung and Veryovka [7]. The utilization of configurable and programmable graphics hardware for real-time hatching and halftoning has been addressed by several researchers [6, 32]. Secord et al. have presented a method for the high-quality distribution of monochrome drawing primitives based on a probability density function derived from an input image [27]. An application of hatching styles to 3D scans of real world enviroments was developed by Xu and Chen [33]. Strothotte and Schlechtweg describe a method for procedural halftoning, which is used by the algorithm presented in this paper [28].

Some researchers have addressed the problem of visualizing hidden components of a graphical model. Nooruddin and Turk have described a preprocessing method for classifying interior and exterior parts of a polygonal dataset [19]. Diepstraten et al. have presented an algorithm for displaying hidden structures in technical illustrations using transparency [3]. An algorithm for the illustrative display of polygonal models using depth peeling has been described by Nienhaus and Döllner [18]. However, their method extracts consecutive layers of geometry for the entire model, again introducing a certain degree of visual complexity. Moreover, their algorithm typically delivers less than interactive frame rates, whereas our new method can generate images in real-time for most datasets.

Non-photorealism has also been applied to direct volume rendering in order to emphasize structures of interest. Rheingans and

Ebert have proposed the *volume illustration* approach for enhancing important features in a volume dataset [23]. A technique that uses stippling for the visualization of volume data has been presented by Lu et al. [14]. Viola et al. describe an automatic method for cutting away irrelevant parts of a volume which occlude significant structures [30]. A method for the real-time generation of line drawings from volume data has recently been presented by Burns et al. [1].

The display of anatomical datasets is one of the most important applications for both direct and indirect volume rendering. An overview of techniques for non-photorealistic medical visualization is given by Preim et al. [22]. Tietjen et al. have presented a combination of different rendering techniques for different parts of a patient dataset [29]. Salah et al. have proposed a technique based on point-based silhouettes and halftoning for a comprehensible display of segmented anatomical data [26].

3 Description of the Algorithm

Our new illustrative rendering algorithm consists of a pipeline of geometry drawing passes followed by an image-space processing step. At first, the polygonal geometry of the iso-surface is rendered twice in order to extract the first and second layers of the model. Subsequently, the optional secondary geometry is rendered, if it has been defined by the user. Finally, an image processing shader combines all data collected in the previous stages to achieve the desired illustrative output. This is the main step of our method. In the following, we denote the set of polygons comprising the iso-surface as **P** and the secondary geometry as **S**.

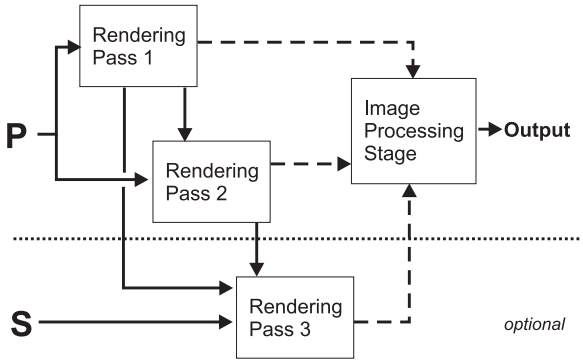


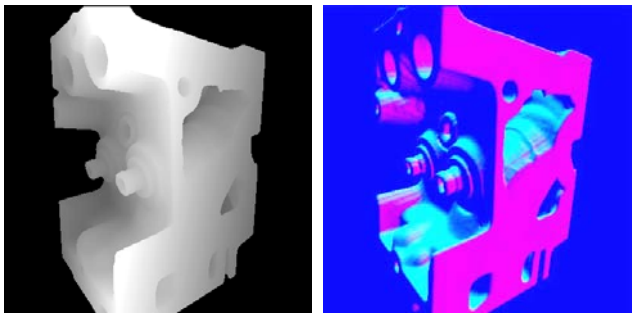
Figure 2: Overview of our algorithm for the illustrative rendering of hidden iso-surface structures. Solid arrows indicate data that is fed into the geometry rendering passes, dotted arrows data for the image processing stage. **P** and **S** stand for the primary iso-surface and the secondary geometry, respectively.

Figure 2 shows an overview of our method. After each of the geometry rendering passes, generated image-space data like fragment depth, normal vectors, and computed intensity are gathered. Note that the second and third rendering step take information from previous passes into account for their own computations. This is necessary because the fragment depths of the polygons drawn by the preceding stages are required for correctly rendering the geometry data in subsequent passes.

In the remainder of this section, the two primary rendering passes (Sec. 3.1) and the step for the optional secondary geometry (Sec. 3.2) are described. Finally, the central image processing stage is discussed in detail in Section 3.3.

3.1 Iso-Surface Rendering Passes

In the first pass of the algorithm, the polygons **P** of the iso-surface are rendered. The standard Z-Buffer test is applied, so that the final image contains the front layer of the iso-surface geometry. For the generation of the image, a special shader is used, which computes the interpolated and normalized normal vectors for each pixel instead of color values. The depth values $depth_{P_1}$ and normal vectors $normal_{P_1}$ are the image-space output of this pass and can be accessed by the following steps of the algorithm. Figure 3 shows the output generated by the first rendering pass for a view of the *Engine* dataset.



(a) $depth_{P_1}$ (b) $normal_{P_1}$

Figure 3: Image-space data generated for the first layer of the *Engine* dataset. The components of the normal vectors are represented as RGB-values in Fig. 3(b).

In the next step of the rendering pipeline, the iso-surface polyons **P** are drawn once again. This time, we want to generate the image that results from removing the first layer of the geometry. The depth information from the first pass is used to achieve this effect using a technique called depth peeling [18]. Each fragment rasterized in this rendering step has to pass two depth tests. In the first test, the depth of the currently regarded polygon fragment is compared to the depth of the first-pass geometry at the same location. The new fragment has to be at a greater depth than the value stored in $depth_{P_1}$, otherwise it is culled. The result of this test is that the first layer of the iso-surface geometry is suppressed, or "peeled away". Subsequently, each fragment has to pass the standard Z-Buffer test so that a coherent second-layer image is generated. This process is illustrated in the following piece of pseudocode:

```
for all polygons p ∈ P do
   F := rasterize p;
   for all fragments f ∈ F do
      /* depth peeling test */
      if (f.depth <= depth_P1(f.x,f.y)) then
         continue;
      endif
      /* standard Z-Buffer test */
      if (f.depth > depth_P2(f.x,f.y)) then
         continue;
      endif
      depth_P2(f.x,f.y) := f.depth;
      normal_P2(f.x,f.y) := f.normal;
   done
done
```

Only fragments which pass both tests contribute to the image-space output of the second rendering stage. The diagram in Figure 4 demonstrates the depth peeling technique. As shown here, the second rendering pass yields the first layer of geometry behind the directly visible polyons.

The output of the second rendering pass again consists of depth and normal information, $depth_{P_2}$ and $normal_{P_2}$. Figure 5 depicts the data computed for the second layer of the *Engine* volume dataset.

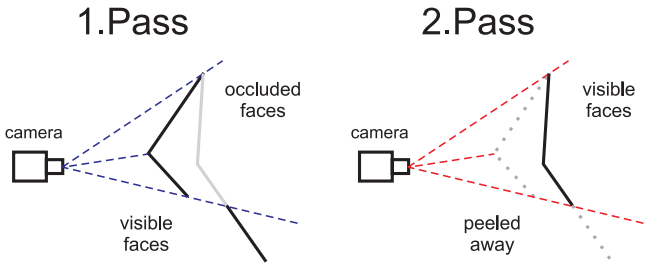
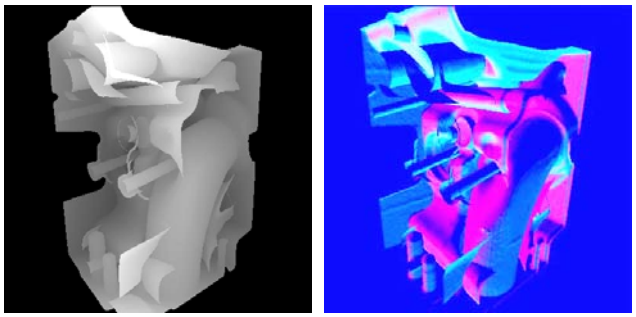


Figure 4: Illustration of the depth peeling technique (adapted from [18]). Thick black lines indicate polygons which are visible in the image generated by the respective rendering stage.



(a) $depth_{P_2}$ (b) $normal_{P_2}$

Figure 5: Image-space data generated for the second layer of the *Engine* dataset.

3.2 Optional Rendering of Secondary Geometry

If a polygonal dataset containing secondary geometry has been specified by the user, the optional third geometry rendering pass is performed. In contrast to the two iso-surface rendering steps, this pass generates an intensity texture. The computed intensities are later used by the image processing stage as brightness values for the secondary geometry pixels. All polygons in **S** are rasterized. Initially, the intensity of each fragment is calculated using diffuse reflection:

$$I_S^0(x,y) = max(normal_S(x,y) \cdot \mathbf{lightDir}, 0) \tag{1}$$

As shown in Equation 1, the initial intensity value I_S^0 is the result of the dot product of the normal vector of the fragment, $normal_S(x,y)$, and the user-defined light direction **lightDir**. A special depth attenuation scheme is then applied to the fragment intensities. The aim of this process is to make the depth relationships in the generated image better understandable. Although the secondary geometry is supposed to remain always visible, we want the user to be able to comprehend its distance relative to iso-surface structures. Therefore, the fragment depth of the secondary geometry is compared to the depth values retrieved from the first two rendering passes.

$$I_S(x,y) = I_S^0(x,y) \cdot \begin{cases} 1, & depth_S(x,y) < depth_{P_1}(x,y) \\ \alpha, & depth_{P_1}(x,y) \le depth_S(x,y) < depth_{P_2}(x,y) \\ \beta, & depth_{P_2}(x,y) \le depth_S(x,y) \end{cases}$$

$$\text{with } 0 < \beta < \alpha < 1 \tag{2}$$

The depth of the secondary geometry fragments is denoted as $depth_S$ in Equation 2. Depth values computed during the primary rendering passes are contained in $depth_{P_1}$ and $depth_{P_2}$, as described in Section 3.1. The following rule determines the attenuation of the initial fragment intensity I_S^0: Secondary geometry fragments which are directly visible, i.e. not behind any iso-surface layer, retain their full intensity. Fragments which would be occluded by the front layer are attenuated with factor α, those which are also behind the second iso-surface layer with factor β. These factors have to be in the interval $[0;1]$ and are parameters of our algorithm.

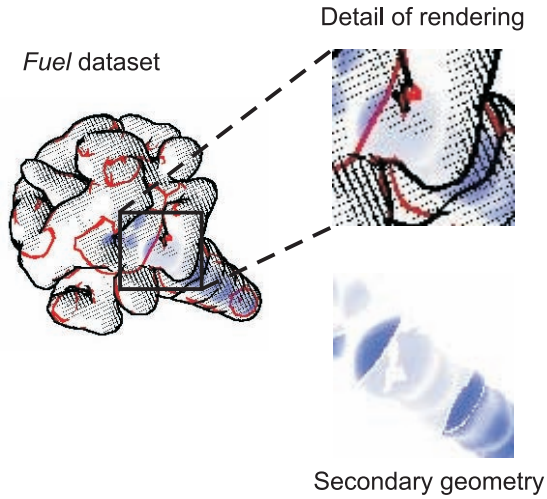


Figure 6: Increasing attenuation of secondary geometry pixels which are behind one or two iso-surface layers, respectively.

This depth attenuation method helps the user understand the spatial relationship between the secondary geometry and the iso-surface. Figure 6 illustrates this effect. In the central part of the image detail, the secondary geometry is shown with less intensity because here it is occluded by two iso-surface layers. The output of the optional third rendering step is the final intensity value I_S, which is accessed by the image processing stage. Moreover, the depth of the secondary geometry fragments, $depth_S$, is also stored for use by the final step of the rendering pipeline.

3.3 Image Processing Stage

All of the aforementioned geometry rendering passes generate images with the full resolution of the OpenGL output window. The size of this OpenGL viewport is determined by the user. In the final stage of our algorithm, all the previously generated data are combined using an image processing step. This step is performed by drawing a textured rectangle, which again has the same size as the OpenGL window. The intermediate images generated by the geometry rendering passes are used as input textures for a special shader program. In this shader, a number of image processing tasks are performed, which eventually yield the desired output rendering.

3.3.1 Silhouette Detection

As mentioned above, the display of silhouettes is a central feature of our illustrative rendering method. Silhouettes are detected in image-space for both the first and the second layer of the iso-surface. Our silhouette detection method is similar to the approach described by Mitchell et al. [16]. Discontinuities in the depth and normal vector images are the basis for computing the response of the silhouette detection filter.

The gradient magnitude is computed for the depth maps of both iso-surface layers, $depth_{P_1}$ and $depth_{P_2}$. In order to determine the gradient magnitude, partial derivatives $\frac{\partial depth_{P_n}}{\partial x}$ and $\frac{\partial depth_{P_n}}{\partial y}$ ($n \in \{1,2\}$) are obtained using the Sobel edge detection filter. The norms of the resulting gradient vectors, $|\nabla depth_{P_1}|$ and $|\nabla depth_{P_2}|$, indicate discontinuities in the depth images. We determine a binary response for the depth discontinuity filter based on a user-defined threshold, *depthThresh*, as shown in Equation 3.

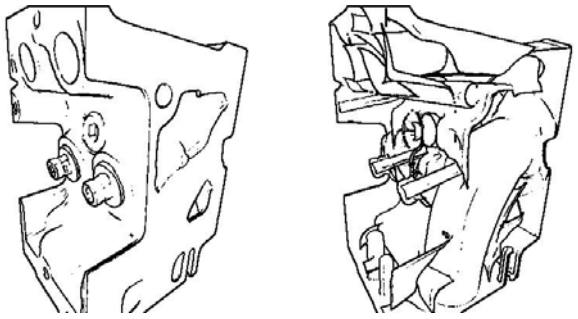
$$depthResp_{P_n}(x,y) = \begin{cases} 0, & |\nabla depth_{P_n}|(x,y) < depthThresh \\ 1, & |\nabla depth_{P_n}|(x,y) \geq depthThresh \end{cases} \quad (3)$$

For finding discontinuities in the normal images, the normal vector stored in each pixel is compared to its four direct neighbours. This comparison is computed as a dot product between the corresponding vectors. As shown in Equation 4, the four resulting dot products are added up in order to obtain a normal vector similarity value $normalSim_{P_n}$. These computations are performed for both layers of the iso-surface, yielding $normalSim_{P_1}$ and $normalSim_{P_2}$, respectively. A smaller normal similarity value indicates a significant discontinuity.

$$\begin{aligned} normalSim_{P_n}(x,y) \;=\; & normal_{P_n}(x,y) \cdot normal_{P_n}(x+1,y) + \\ & normal_{P_n}(x,y) \cdot normal_{P_n}(x-1,y) + \\ & normal_{P_n}(x,y) \cdot normal_{P_n}(x,y+1) + \\ & normal_{P_n}(x,y) \cdot normal_{P_n}(x,y-1) \end{aligned} \quad (4)$$

$$normalResp_{P_n}(x,y) = \begin{cases} 0, & normalSim_{P_n}(x,y) \geq normalThresh \\ 1, & normalSim_{P_n}(x,y) < normalThresh \end{cases} \quad (5)$$

A binary normal discontinuity response is computed by comparing the similarity value to a user-defined threshold (see Equation 5). The greater the threshold value *normalThresh* is, the more pixels contribute to normal map silhouettes. Finally, an overall silhouette detection response is determined for each iso-surface layer. Depth as well as normal discontinuities are taken into account by calculating the logical OR of both response types, i.e. $silhouette_{P_n}(x,y) = depthResp_{P_n}(x,y) \vee normalResp_{P_n}(x,y)$. The output of the silhouette detection process for both layers of the *Engine* dataset is shown in Figure 7.



(a) $silhouette_{P_1}$ (b) $silhouette_{P_2}$

Figure 7: Silhouette detection responses computed for the first and second layer of the *Engine* dataset iso-surface.

3.3.2 Procedural Halftoning

In our illustrative rendering style, the shape of the front layer of the iso-surface is not only represented with silhouettes. In order to make a better understanding of the geometry of the iso-surface possible, and to achieve a clear distinction between first-layer and second-layer structures, diffuse reflection intensities are added to the graphical output. The intensity of each first-layer fragment is computed as the dot product of a user-defined light direction and the stored normal vector. As shown in Equation 6, we again use the aforementioned light direction vector **lightDir** for this calculation.

$$I_{P_1}(x,y) = max(normal_{P_1}(x,y) \cdot \mathbf{lightDir}, 0) \quad (6)$$

The main aim of our rendering algorithm is the display of hidden structures behind the first iso-surface layer. We therefore want to show the diffuse reflection intensities with a method which allows second-layer silhouettes to remain visible. A monochrome halftoning method is used for recreating a black-and-white cross-hatching style found in technical illustrations. We utilize the procedural screening approach described by Strothotte and Schlechtweg [28], which does not require additional input textures for describing the hatching pattern.

The coordinates of each fragment which is to be dithered are first mapped to dither coordinates (s,t). This is done using a mapping function $\mathbf{M}$:

$$\begin{aligned} (x',y') &= R_\theta \cdot (x,y) \\ (s,t) &= \mathbf{M}(x',y') = \left(\tfrac{x' \bmod n}{n}, \tfrac{y' \bmod n}{n} \right) \end{aligned} \quad (7)$$

In Equation 7, the screen-space coordinates of the currently regarded fragment are denoted as (x,y). These coordinates are first rotated by the angle θ, which is selected by the user. This is done by computing the product of the rotation matrix $R_\theta \in \Re^{2x2}$ and the input coordinate vector. Due to this multiplication, the resulting halftoning pattern will be rotated relative to the screen-space coordinate axes. This creates a more natural look of the black-and-white hatching image. Subsequently, the rotated fragment coordinates are mapped to the real number range $[0;1]$ using the mapping function $\mathbf{M}$. The dither coordinates (s,t) are the basis for determining whether the fragment is displayed as an opaque black pixel or as translucent. The choice of the user-defined variable n in Equation 7 determines the size in pixels of the hatching pattern.

$$\tau(s,t) = \begin{cases} c_{cross} \cdot t & s \leq c_{cross} \\ (1 - c_{cross})s + c_{cross} & s > c_{cross} \end{cases} \quad (8)$$

$$I_{hatching}(x,y) = \begin{cases} 0, & I_{P_1}(x,y) < \tau(s,t) \\ 1, & I_{P_1}(x,y) \geq \tau(s,t) \end{cases} \quad (9)$$

For each pixel, a halftoning threshold τ is computed as a function of the dither coordinates (s,t). The calculation of τ depends on the parameter c_{cross} (see Equation 8). c_{cross} determines the minimum intensity necessary for generating perpendicular cross-hatching strokes in addition to the parallel strokes which are the basis for the halftoning pattern. As described in Equation 9, the diffuse reflection intensity $I_{P_1}(x,y)$ is then compared to the local threshold $\tau(s,t)$. If the intensity is large enough, the output value $I_{hatching}(x,y)$ will be one, otherwise zero. The black-and-white pattern computed by this halftoning method for the intensity range $[0;1]$ is depicted in Figure 8.

3.3.3 Display of Secondary Geometry

In the final output image, the secondary geometry is displayed as a solid polygonal object with a special color. The brightness of secondary geometry fragments is determined by the intensity value $I_S(x,y)$, which has been computed in the optional third geometry rendering pass (see Section 3.2).

Figure 8: Black-and-white pattern generated for the continuous intensity range $[0;1]$.

Before the final color for each pixel in the output image is computed, a specific depth test is performed for the secondary geometry. In places where second-layer silhouette pixels have been detected, only such secondary geometry fragments are to be displayed which are in front of the second layer of the iso-surface.

$$I'_S(x,y) = I_S(x,y) \cdot \begin{cases} 0, \; silhouette_{P_2}(x,y) \wedge (depth_{P_2}(x,y) \leq depth_S(x,y)) \\ 1, \; otherwise \end{cases} \tag{10}$$

Equation 10 shows the additional depth test for secondary geometry pixels, which yields the final intensity value I'_S. Due to this test, the depth relationships between second-layer silhouettes and the secondary geometry are correctly represented.

3.3.4 Computation of Final Pixel Color

In order to generate the final output image, all the data computed for each pixel so far are combined. The image is initialized with a user-defined background color called *paperColor*. We typically use a bright background color to create the look of a technical illustration on paper. The second-layer silhouettes are then drawn over the background. They are displayed with the color *backLayerColor* in all places where $silhouette_{P_2}$ indicates a silhouette fragment. Subsequently, the secondary geometry is blended over the background according to its computed intensity I'_S. The color of the secondary geometry is also selected by the user and stored in the variable *secGeomColor*. Finally, the front-layer of the iso-surface is taken into account. Every pixel which has a first-layer silhouette response of one or a hatching intensity of zero is displayed black. In this paper, we always use black for the first-layer geometry, but it could easily be replaced with any other color. The following piece of pseudocode summarizes the process of determining the final pixel color. This code uses a terminology similar to the functionality of modern shading languages.

```
vec3 pixelColor;
pixelColor = mix(paperColor, backLayerColor, silhouette_P2(x,y));
pixelColor = mix(pixelColor, secGeomColor, I'_S(x,y));
pixelColor *= (1.0 - silhouette_P1(x,y));
pixelColor *= I_hatching(x,y);
```

In this pseudocode, the variable `pixelColor` is an RGB color vector, which will store the final output color of the pixel at position (x,y). In the code, the linear blend of two colors is computed by the function `mix`, i.e., `mix`(c_1, c_2, α)=$(1-\alpha)c_1 + \alpha c_2$. The black color of the first-layer geometry is generated by multiplying the components of `pixelColor` with a scalar value of zero, indicated by an `*=` operator.

4 Implementation Details

The capability for achieving real-time frame rates has been one of the main objectives of the design and the implementation of our algorithm. Our method has been realized with the OpenGL Shading Language [24], which was used to implement all of the aforementioned shader programs. The entire approach is executed on the graphics processing unit (GPU) and does not require any preprocessing on the CPU.

Each type of intermediate image-space information (e.g., $depth_{P_n}$, $normal_{P_n}$, $depth_S$ etc.) is stored in a separate texture image in onboard memory. The normal and intensity images are generated as the components of standard RGBA textures, while $depth_{P_n}$ and $depth_S$ have a special depth texture format provided by OpenGL. In each geometry rendering pass, the intermediate data are written into the framebuffer. Subsequently, they are copied into texture images which are accessed by later stages of the algorithm. This is done with the `glCopyTexSubImage2D()`[1] function.

In the final step of the algorithm, a rectangle covering the entire OpenGL window is drawn using the image processing shader. The image processing shader reads the intermediate data by accessing the corresponding texture images. In particular for the silhouette detection step, a large number of texture accesses is required. For each fragment, a neighborhood of five or nine texels has to be read for the normal and depth textures, respectively. In order to avoid a loss of performance due to the repeated calculation of texel addresses, we use a special precomputation scheme. This texel address precomputation method is similar to the one presented by Viola et al. [31].

One significant advantage of the design of our rendering pipeline (see Fig. 2) is the fact that all image processing tasks are performed in the same shader progam. This way, no redundant texture accesses are necessary. All the previously computed data relevant for the current pixel are loaded once from the intermediate textures and can be used for several computations. One example is the first-layer normal information, $normal_{P_1}$, which is needed for the silhouette detection and the calculation of the diffuse reflection intensity.

Many of the computations in our rendering pipeline use standard functions provided by the shading language. Examples include `step()`, which performs a boundary check, and `mix()`, which computes a linear blending of vectors. These functions are normally executed efficiently on the GPU. Due to the extensive use of these functions, our implementation requires almost no conditional branches and no loop statements on the GPU, which are notoriously slow.

5 Results

We have tested the algorithm with a number of example datasets. Most of them are polygon meshes representing iso-surfaces generated from volume data. For some of the datasets, secondary geometry was created by extracting a second surface with a different iso-value. Example images rendered with our method are shown in Figures 1, 11 and 12. The *Engine* dataset (Fig. 1(a)) is a CT scan of two cylinders of an engine block. The result of a simulation of fuel injection into a combustion chamber is contained in the *Fuel* dataset (Fig. 1(b)). In this case, high-intensity voxels are displayed as secondary geometry. The *NegHip* example (Fig. 12(a)) is a simulation of the spatial probability of the electrons in a protein molecule. It also contains secondary geometry extracted from high-intensity voxels. A scanned human colon is shown in the *Colon* example, and the *Ventricle* dataset is a scan of the ventricular system of a human brain (Fig. 12(b) and 12(c)). In addition to these iso-surfaces, we have also tested our algorithm with a manually created polygonal model. The *Screwdriver* dataset (Fig. 11) is the detailed mechanical design of an electric screwdriver, with some inner parts highlighted as secondary geometry.

The example images and the animations in the accompanying

[1] An alternative to using `glCopyTexSubImage2D()` is the direct rendering of intermediate images into texture memory. We have implemented a modification of our algorithm with render-to-texture using the OpenGL framebuffer object extension (`EXT_framebuffer_object`). However, we have found the resulting performance gains to be only marginal, due to the relatively limited amount of image-space data which needs to be copied. The performance measurments in this paper are based on the original approach with explicit texture copying.

video show that our method can convey the shape of hidden structures of the iso-surface. This is also illustrated in Figure 9, which shows a detail of the colon dataset, where the shape of the back wall of the colon is suggested by second-layer silhouettes. In Figure 10, holes in the septum of the patient are visible in the *Ventricle* dataset. These are defects caused by a degenerative process, and are significant for medical diagnosis and treatment.

Figure 9: Close-up of a section of the *Colon* dataset.

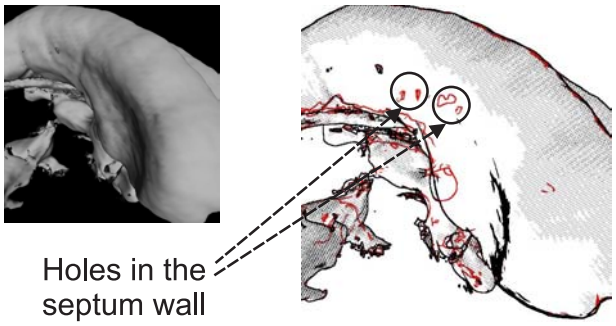


Figure 10: A significant detail in the *Ventricle* dataset displayed by our algorithm.

We have run a number of tests for measuring the frame rates achieved with our new method. These benchmarks were performed on a computer with a Pentium 4 processor running at 2.8 GHz using a graphics card with an NVidia GeForce FX 6600 GT chipset. Animation sequences with a length of at least 1000 frames were used to compute the average performance for each test run. Table 1 shows a comparison of frame rates measured with standard rendering and with our illustrative method for the *Engine* dataset.

Table 1: Comparison of frame rates for standard and illustrative rendering of the *Engine* dataset (311k vertices) at different resolutions.

	1024x768	**800x600**	**640x480**
standard	96.01	99.05	100.85
illustrative	21.33	28.10	34.83

In Table 2, rendering speeds achieved by our algorithm for the other five test datasets are listed. Benchmark runs were performed for different output resolutions. As illustrated in both benchmark tables, the performance of our method depends on two main factors. The first is the size of the iso-surface mesh. A large number of polygons slows down the image generation because of the multiple geometry rendering passes. Moreover, the frame rate is influenced strongly by the output resolution. A larger output window increases the size of the intermediate textures and the number of image processing operations, resulting in a reduced rendering speed. Still, our algorithm is capable of delivering real-time performance in most cases. Even for large datasets and image resolutions, frame rates of close to or above 20 fps have been measured.

Table 2: Frame rates measured for the other five datasets.

Dataset	**#vertices**	**1024x768**	**800x600**	**640x480**
Colon	1,076k	18.16	22.96	27.17
Screwdriver	487k	26.65	40.69	58.07
Ventricle	201k	31.00	45.94	61.98
NegHip	17k	30.96	50.41	76.78
Fuel	6.4k	32.29	52.30	81.49

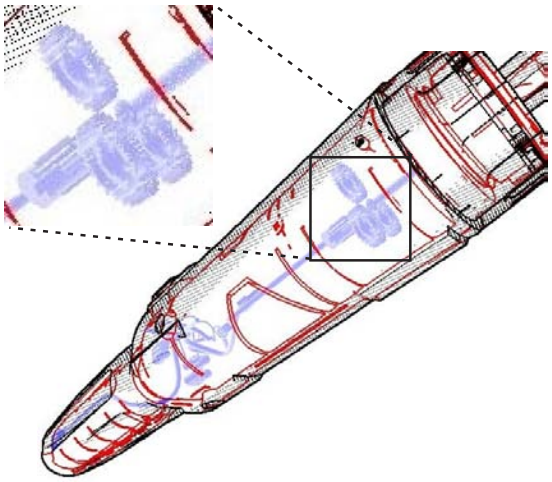


Figure 11: Detailed polygonal model of an electric screwdriver. Some inner parts are used as secondary geometry (see enlarged detail).

6 Conclusion

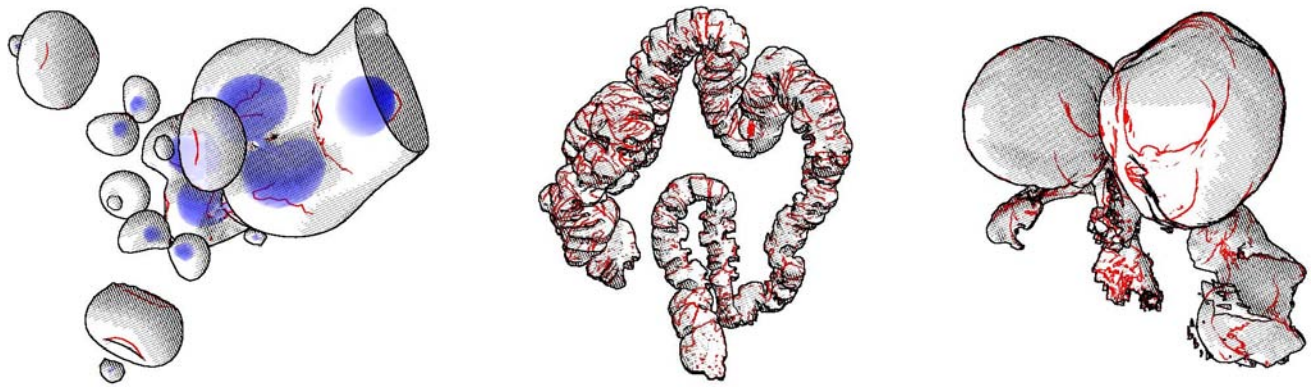
We have presented a novel method for the illustrative rendering of iso-surfaces. Our algorithm utilizes depth peeling, silhouette extraction, and monochrome hatching and combines them in an efficient way in a single rendering pipeline. The generated images make a simultaneous inspection of the outer surface and inner structures possible. Due to the selected style of rendering, spatial relationships and the shape of hidden objects can easily be understood.

One major advantage of our algorithm is the fact that no preprocessing of the input data is necessary. Any polygonal dataset can be loaded into the system and then be displayed in the new illustrative style. The depth peeling step automatically extracts the second-layer geometry. Since we always assume the second iso-surface layer to contain the relevant hidden structures, difficulties can arise in the case of more complex datasets. If areas of interest are occluded by several layers of polygons, they have to be manually specified as secondary geometry in order to remain visible. However, the continual display of such secondary structures is integrated efficiently into our rendering pipeline.

Although the performance of our algorithm depends on the size of the displayed dataset and the image resolution, we have found it to achieve real-time frame rates in most cases.

Acknowledgments

We would like to thank Ángel del Río for providing the iso-surface extraction software used for generating our example datasets from volume data. Most of the volume datasets used in our experiments were downloaded from the VolVis website (*www.volvis.org*). This work has been supported by project VIRTUE in the focus program on "Medical Navigation and Robotics" (SPP 1124) of the German Research Foundation (DFG).



(a) *NegHip* dataset (b) *Colon* dataset (c) *Ventricle* dataset

Figure 12: Example images generated with our illustrative rendering algorithm. (Parameters used for all images: $\alpha = 0.7$, $\beta = 0.21$, $normalThresh = 3.98$, $depthThresh = 0.001$, $\theta = 20.0$, $n = 4.0$, $c_{cross} = 0.9$)

REFERENCES

[1] M. Burns, J. Klawe, S. Rusinkiewicz, A. Finkelstein, and D. DeCarlo. Line Drawings from Volume Data. In *Proceedings of ACM SIGGRAPH*, 2005.

[2] P. Decaudin. Cartoon-Looking Rendering of 3D-Scenes. Research Report 2919, INRIA, June 1996.

[3] J. Diepstraten, D. Weiskopf, and T. Ertl. Transparency in Interactive Technical Illustrations. In *Proceedings of Eurographics*, 2002.

[4] M. Eißele, D. Weiskopf, and T. Ertl. The G2-Buffer Framework. In *Simulation and Visualization*, pages 287–298, 2004.

[5] J.A. Ferwerda. Three Varieties of Realism in Computer Graphics. In *Proceedings SPIE Human Vision and Electronic Imaging*, pages 290–297, 2003.

[6] B. Freudenberg, M. Masuch, and T. Strothotte. Real-Time Halftoning: A Primitive for Non-Photorealistic Shading. In *Rendering Techniques 2002: Proceedings of the 13th Eurographics Workshop on Rendering*, pages 227–231, June 2002.

[7] J. Fung and O. Veryovka. Pen-and-ink Textures for Real-Time Rendering. In *Proceedings of Graphics Interface*, 2003.

[8] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A Non-Photorealistic Lighting Model For Automatic Technical Illustration. In *Proceedings of ACM SIGGRAPH*, 1998.

[9] B. Gooch and A. Gooch. *Non-Photorealistic Rendering*. A K Peters, 2nd edition, 2001.

[10] A. Hertzmann and D. Zorin. Illustrating Smooth Surfaces. In *Proceedings of ACM SIGGRAPH*, pages 517–526, 2000.

[11] V. Interrante, H. Fuchs, and S. Pizer. Conveying the 3D Shape of Smoothly Curving Transparent Surfaces via Texture. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):98–117, March 1997.

[12] S. Kim, H. Hagh-Shenas, and V. Interrante. Conveying Shape with Texture: experimental investigations of texture's effects on shape categorization judgments. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):471–483, 2004.

[13] W.E. Lorensen and H.E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169, July 1987.

[14] A. Lu, C.J. Morris, D.S. Ebert, P. Rheingans, and C. Hansen. Non-Photorealistic Volume Rendering Using Stippling Techniques. In *Proceedings of IEEE Visualization*, 2002.

[15] E.B. Lum and K.-L. Ma. Non-Photorealistic Rendering Using Watercolor Inspired Textures and Illumination. In *Ninth Pacific Conference on Computer Graphics and Applications*, page 322, 2001.

[16] J. L. Mitchell, C. Brennan, and D. Card. Real-Time Image Space Outlining for Non-Photorealistic Rendering. In *SIGGRAPH 2002 Conference Abstracts and Applications*, page 239, 2002.

[17] M. Nienhaus and J. Döllner. Edge-Enhancement - An Algorithm for Real-Time Non-Photorealistic Rendering. In *Proceedings of WSCG*, pages 346–353, 2003.

[18] M. Nienhaus and J. Döllner. Blueprints - Illustrating Architecture and Technical Parts using Hardware-Accelerated Non-Photorealistic Rendering. In *Proceedings of Graphics Interface*, 2004.

[19] F.S. Nooruddin and G. Turk. Interior/Exterior Classification of Polygonal Models. In *Proceedings of IEEE Visualization*, pages 415–422, 2000.

[20] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. Avila, R. Machiraju, and J. Lee. The Transfer Function Bake-off. *Computer Graphics and Applications*, 21(3):16–22, 2001.

[21] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-Time Hatching. In *Proceedings of ACM SIGGRAPH*, page 581, 2001.

[22] B. Preim, C. Tietjen, and C. Doerge. Npr, Focussing and Emphasis in Medical Visualizations. In *Proceedings of Simulation und Visualisierung*, 2005.

[23] P. Rheingans and D. Ebert. Volume Illustration: Nonphotorealistic Rendering of Volume Models. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):253–264, July-Sept 2001.

[24] R.J. Rost. *OpenGL Shading Language*. Addison-Wesley Publishing Company, 2004.

[25] T. Saito and T. Takahashi. Comprehensible Rendering of 3-D Shapes. In *Proceedings of ACM SIGGRAPH*, pages 197–206, 1990.

[26] Z. Salah, D. Bartz, and W. Straßer. Illustrative Rendering of Segmented Anatomical Data. In *Proceedings of Simulation und Visualisierung*, 2005.

[27] A. Secord, W. Heidrich, and L. Streit. Fast Primitive Distribution for Illustration. In *Proceedings of Eurographics Workshop on Rendering*, 2002.

[28] T. Strothotte and S. Schlechtweg. *Non-Photorealistic Computer Graphics - Modelling, Rendering, and Animation*. Morgan Kaufmann Publishers, 2002.

[29] C. Tietjen, T. Isenberg, and B. Preim. Combining Silhouettes, Surface, and Volume Rendering for Surgery Education and Planning. In *Proceedings of IEEE/Eurographics Symposium on Visualization*, 2005.

[30] I. Viola, A. Kanitsar, and E. Gröller. Importance-Driven Volume Rendering. In *Proceedings of IEEE Visualization*, pages 139–145, 2004.

[31] I. Viola, A. Kanitsar, and M.E. Gröller. Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages. In *Proceedings of IEEE Visualization*, pages 309–316, 2003.

[32] M. Webb, E. Praun, A. Finkelstein, and H. Hoppe. Fine Tone Control in Hardware Hatching. In *Second International Symposium on Non Photorealistic Rendering*, pages 53–58, June 2002.

[33] H. Xu and B. Chen. Stylized Rendering of 3D Scanned Real World Environments. In *Proceedings of the 3rd International Symposium on Non-photorealistic Animation and Rendering*, pages 25–34, 2004.