EUROGRAPHICS 2003 / P. Brunet and D. Fellner
(Guest Editors)

Volume 22 (2003), Number 3

Interactive Cutaway Illustrations

J. Diepstraten D. Weiskopf T. Ertl

Institute for Visualization and Interactive Systems,
University of Stuttgart, Germany

Abstract

In this paper we discuss different approaches to generate cutaway illustrations. The purpose of such a drawing is to allow the viewer to have a look into an otherwise solid opaque object. Traditional methods to draw these kinds of illustrations are evaluated to extract a small and effective set of rules for a computer-based rendering of cutaway illustrations. We show that our approaches are not limited to a specific rendering style but can be successfully combined with a great variety of well-known artistic or technical illustration techniques. All methods of this paper make use of modern graphics hardware functionality to achieve interactive frame rates.

1. Introduction

Research in the area of non-photorealistic rendering (NPR) has grown a lot over the past few years, especially in the field of illustrations and artistic rendering. Technical illustrations are of particular interest in NPR research as they cover a quite large field of applications. Today technical illustration styles can be found in manuals, text and science books, advertisement, or even computer games[18]. One of the major advantages of technical illustrations in contrast to photorealistic renderings or actual photographs is that they can provide a selective view on important details while extraneous details are omitted[6]. For example, NPR styles may be used to improve the recognition of the shape and structure of objects, their orientation, or spatial relationships.

Research on automatically generating technical illustrations has been focused on imitating the different rendering styles traditionally used by illustrators. Unfortunately, these computer-based techniques only provide some important details like shape, structure, or depth information, but often neglect complex spatial relationships and especially issues of occlusion. In photorealistic rendering, spatial relationships between objects can be shown by using transparency. This is also possible with technical illustrations and, in fact, this solution is sometimes used by illustrators.

However, illustrators often prefer to use cutaway techniques. Cutaway drawings in technical illustrations allow the user to view the interior of a solid opaque object. In these illustrations, entities lying inside or going through an opaque object are of more interest than the surrounding one itself. Instead of letting the inner object shine through the girdling surface, parts of the exterior object are removed. This produces a visual appearance as if someone had cutout a piece of the object or sliced it into parts. Cutaway illustrations avoid ambiguities with respect to spatial ordering, provide a sharp contrast between foreground and background objects, and facilitate a good understanding of spatial ordering. Another reason for the popularity of cutaway illustrations might be the fact that the appearance of semi-transparent surfaces is hard to simulate with most classical drawing styles for hand-made illustrations.

The purpose of this paper is to provide methods to generate cutaway drawings on a computer. These methods are based on a small and effective set of rules that are extracted from traditional techniques. Even though our rules lead to a completely automatic generation of quite reasonable initial cutaway illustrations, interactivity is still very useful for a fine adjustment of the initial parameters. Therefore, we present ways to map the cutaway renderings directly to modern graphics hardware in order to achieve interactive frame rates. Another aspect of this work is to show that cutaway is de-coupled from the rendering style used in the final image. For this reason the cutaway techniques can readily be included in a great variety of already existing rendering systems.

2. Previous and Related Work

To our knowledge, the SIGGRAPH 99 advanced OpenGL rendering course[2] is the only work in the field of computer graphics research that explicitly mentions cutaway illustration. In this course, a simple method using α blending is discussed to achieve an appearance of cutaway drawings. However, α blending causes rather a smearing out than a cutting out or slicing. Other work deals with using transparency in the context of NPR to show inner-space relationships[7, 14, 5].

Most research on technical illustrations has been focused on simulating different rendering styles. Gooch et al.[11] present tone-based shading and various silhouette rendering methods. Dooley and Cohen[6], Winkenbach et al.[27], and Salisbury et al.[23] investigate pen-and-ink illustrations. Recent work by Raskar[22], Praun et al.[20], and Freudenberg et al.[10] is based on hardware capabilities to render different rendering styles in real time. We demonstrate that these rendering approaches are independent of our cutaway techniques and that cutaway can be readily combined with these rendering styles.

Another related field of research on technical illustrations deals with rendering on a higher level of abstraction, where semantics and user interaction have to be taken into account. An important aspect is labeling and annotating illustrations, cf., for example, the recent work by Bourguignon et al.[3]. Seligman and Feiner introduce a rule-based illustration system[24] for rendering photorealistic illustrations, with an extension for supporting interactivity[25].

3. Overview of Cutaway Illustrations

In this section we briefly review how *cutaway* illustrations are traditionally created by illustrators in order to extract some requirements for an automatic generation process on the computer. For detailed background information on hand-made technical illustrations in general and cutaway drawings in particular we refer to classical artbooks[26, 17].

The purpose of a cutaway drawing is to allow the viewer to have a look into an otherwise solid opaque object. Instead of letting the inner object shine through the surrounding surface, parts of outside object are simply removed. From an algorithmic point of view, the most interesting question is where to cut the outside object. The answer to this fundamental question depends on many different factors, for example, the sizes and shapes of the inside and outside objects, the semantics of the objects, personal taste, etc. Many of these factors cannot be formalized in the form of a simple algorithm and need some user interaction. Nevertheless, we found some interesting common properties in many examples of traditional cutaway drawings which allow us to automatically generate quite reasonable cutaways.

In this paper we distinguish between two different subclasses of the general notion of a *cutaway* drawing: *cutout* and *breakaway*. Figures 1 (a) and 1 (b) show the difference between the two subclasses.

Artists and illustrators tend to restrict themselves to very simple and regularly shaped cutout geometries. Often only a small number of planar slices is cut into the outside object; in many cases just two planes are sufficient. The location and orientation of the cutting planes are determined by the spatial distribution of the interior objects and, more importantly, by the geometry of the outside body. Just enough is taken away from the outlying object to allow the observer to view the internal details. We have analyzed many cutaway illustrations and have come to the conclusion that two planes intersecting at an angle between 90 to 140 degrees are sufficient for a wide class of applications. Another common property concerns the location of the slicing planes. The cut through the object of interest often takes place at or around its main axis. The main axis of an object is the axis with the greatest spread.

A cutout in a technical illustration has not always to be smooth. For example, a sawtooth-like or jittering cutting is often applied to better distinguish between outer and inner objects and produce a higher level of abstraction. Figure 1 (a) shows such a jittering cutout for a simple example scene. The image was generated by the computer-based method described in Section 4.

The cutout approach is particularly useful when many objects or large objects are inside and cover a large portion of the interior of the girdling object. In contrast, if only a few small inside objects lie densely to each other, another approach is more appropriate. Here, an illustrator rather breaks a virtual hole into the boundary to show the interior objects. This boundary hole should be just wide enough to see these objects. We call this method *breakaway*. Figure 1 (b) shows a simple example of a breakaway illustration. The image was generated by the computer-based technique described in Section 5.

It has already been stated that hand-made illustrations are influenced by various aspects many of which are hard to be formalized for a computer-based processing. Nevertheless, we have been able to extract a small number of rules that lead to quite convincing, fully automatic cutaway drawings. The user is still able to change parameters after the initial automatic construction.

Let us start with the rules for the cutout approach. The first, very basic question is: Which objects are potentially subject to cutting? We have observed that interior objects are not sliced by the cutout geometry. Cutting is only applied to outside objects. Therefore, the first requirement is:

(R1) Inside and outside objects have to be distinguished from each other.

Please note that this requirement not only covers scenes with a single outside object, but has to allow for scenarios with several disjoint outside objects and even nested layers of outside objects. Another issue is the shape of the cutout geometry. We restrict ourselves to a specific class of shapes:

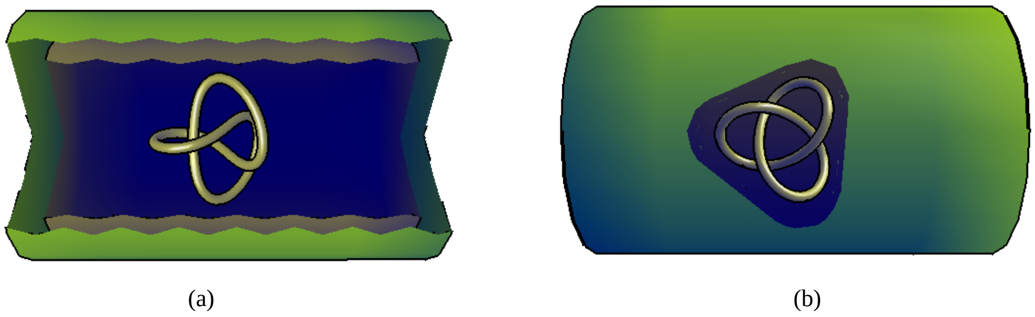

Figure 1: *Comparison of computer-generated cutout and breakaway illustrations. The left image demonstrates the cutout technique with a jittering boundary. In the right picture, the breakaway method is applied to the same scene.*

(R2) The cutout geometry is represented by the intersection of (a few) half spaces[†].

By construction, this cutout geometry is convex. Very often good results are achieved by:

(R2') A cutout geometry is represented by the intersection of two half spaces.

The next rule determines the position and orientation of the above cutout geometry:

(R3) The cutout is located at or around the main axis of the outside object.

For a cutout with two planes, the intersection line between the two planes lies on the main axis. The angle of rotation around this axis is a free parameter that can be adjusted by the user.

A cutout in a technical illustration does not always have to be as smooth as the geometry defined by a collection of half spaces. For example, a sawtooth-like or jittering cutting is often applied to such a simple cutout geometry to produce a higher level of abstraction.

(R4) An optional jittering mechanism is useful to allow for rough cutouts.

Finally, the cutout produces new parts of the objects' surfaces at the sliced walls. This leads to:

(R5) A possibility to make the wall visible is needed.

This requirement is important in the context of boundary representations (BReps) of scene objects, which does not explicity represent the solid interior of walls. Therefore, special care has to be taken to make possible a correct illumination of cutout walls.

The other cutaway approach in the form of a breakaway is based on a slightly different set of rules. The first requirement *(R1)* for a distinction between inside and outside objects is the same as in cutout drawings. However, the shape and position of the breakaway geometry is not based on rules *(R2)* and *(R3)*, but on:

(R6) The breakaway should be realized by a single hole in the outside object.

If several small openings were cut into the outside surface, a rather disturbing and complex visual appearance would be generated. Nevertheless:

(R7) All interior objects should be visible from any given viewing angle.

The above rule for making the walls visible *(R5)* can be applied to breakaway illustrations as well. Jittering breakaway illustrations are seldom and therefore *(R4)* is not a hard requirement for these illustrations.

In the following two sections, we present two different rendering algorithms which meet the above characteristics for cutout and breakaway illustrations, respectively.

4. Cutout Drawings

In this section a class of rendering algorithms for cutout drawings is presented. We show how a computer-based process can fulfill the aforementioned rules *(R1)–(R5)* for the cutout approach.

Classification. We assume that the classification of objects as interior or exterior *(R1)* is provided by an outside mechanism. The problem is that a generic classification criterion solely based on the spatial structure of the scene objects is not available. For special cases, objects can be recognized as being inside or outside by observing their geometry. For example, the important class of nested surfaces can be handled by an algorithm by Nooruddin and Turk[19]. However, this method does not support exterior objects that already have openings before the cutting process is performed.

[†] A half space can be represented by the plane that separates the space from its complement.

In our implementation, a different approach is taken. The objects are stored in a scenegraph structure; the classification is based on an additional Boolean attribute that is attached to each geometry node of the scenegraph. For practical purposes, geometric modeling of scenes is performed in an outside, commercial modeling and animation tool (such as 3D Studio Max or Maya) and the scenes are afterwards imported into our software via a 3D data file. Since these modeling tools do not directly support the additional classification attribute, other information stored in the 3D file format has to be exploited. For example, the transparency value can be (mis-) used, or the classification is coded in the form of a string pattern in the name of 3D objects. This approach allows the user to explicitly specify interior and exterior objects and to introduce some external knowledge into the system.

Main Axis. Another issue is the computation of the main axes of the outside objects. Each object is assumed to be represented by a triangulated surface. Information on the connectivity between triangles is not required, i.e., a "triangle soup" can be used.

The algorithm makes use of first and second order statistics that summarize the vertex coordinates. These are the mean value and the covariance matrix[8]. The algorithm is identical to Gottschalk et al.'s[13] method for creating object-oriented bounding boxes (OBB). If the vertices of the i'th triangle are the points $\vec{p}^i$, $\vec{q}^i$ and $\vec{r}^i$, then the mean value $\vec{\mu}$ and the covariance matrix C can be expressed in vector arithmetics as

$$\vec{\mu} = \frac{1}{3n}\sum_{i=0}^{n}(\vec{p}^i + \vec{q}^i + \vec{r}^i) \quad ,$$

$$C_{jk} = \frac{1}{3n}\sum_{i=0}^{n}(\vec{p}'^i_j \vec{p}'^i_k + \vec{q}'^i_j \vec{q}'^i_k + \vec{r}'^i_j \vec{r}'^i_k) \quad ,$$

where n is the number of triangles, $\vec{p}'^i = \vec{p}^i - \vec{\mu}$, $\vec{q}'^i = \vec{q}^i - \vec{\mu}$, $\vec{r}'^i = \vec{r}^i - \mu$, and C_{jk} are the elements of the 3×3 covariance matrix. Acutally, not the vertices of the original mesh are used, but the vertices of the convex hull. Moreover, a uniform sampling of the convex hull is applied to avoid potential artifacts caused by unevenly distributed sizes of triangles. These improvements are also described by Gottschalk et al.[13] The eigenvectors of a symmetric matrix with different eigenvalues are mutually orthogonal. The eigenvectors of the symmetric covariance matrix can be used as an orthogonal basis. Of special interest is the eigenvector corresponding to the largest eigenvalue because it serves as the main axis of the object.

CSG Cutout. Finally, the cut geometry has to be defined and then applied at the previously determined location. An object-space approach working directly on the geometry could be realized by techniques known from constructive solid geometry (CSG).

The intersection of half spaces *(R2)* can be realized by a CSG intersection operation working on half spaces. An intersection operation is equivalent to a logical "and" applied to the corresponding elements of the spaces. A half space can be represented by the plane that separates the space from its complement. Therefore, this plane serves as a slicing plane in the cutout approach. Note that we define the half space in a way that outside objects are removed at all locations of this half space; outside objects are left untouched in the complementary space. The actual cutting process is modeled by a CSG difference operation applied to the cutout geometry and the geometry of the exterior objects. Intrinsic to all CSG operations is the creation of new boundary surfaces at cuts. Therefore, cutout walls are automatically modeled and can be displayed afterwards, as required by *(R5)*.

In the general approach of rule *(R2)*, the number, locations, and orientations of the cutting planes have to be defined by the user. The more restricted rule *(R2')* prescribes a fixed number of two planes. Moreover, the intersection line between the planes is fixed by the main axis of the outside object. The only free parameters are the relative angle between the planes and the angle of rotation of the cutout geometry with respect to the main axis. The relative angle can be set to a default value in between 90 and 140 degrees (e.g., to 110 degrees); the default orientation with respect to the main axis can be set to any fixed angle. With these initial values, quite good results are achieved without any user interaction.

For an optional sawtooth-like or jittered cutout *(R4)* the cutout geometry has to be perturbed, for example, by a displacement mapping technique[4]. Appropriate kinds of displacement maps are presented shortly in the description of texture-based cutouts.

Although all the requirements *(R1)–(R5)* can be directly mapped to a CSG-based implementation, we have not pursued this approach in more detail. The main problem is that CSG Boolean operations can be very time-consuming. Therefore, parameter changes are unlikely to work in real time and interactive work is not possible. This is a major drawback because cutout drawings—even if they work almost automatically—need some user interaction to adjust parameters for improved final results. If highly detailed and jittered cutout geometries or complex exterior objects are used, CSG operations become particularly time-consuming. Another issue is rendering time itself. Often a high number of new primitives is introduced by the re-tesselating steps required for precise intersections between objects. The high amount of new primitives could hinder interactive rendering times and might require further object optimization steps. All these aspects limit the applicability of the CSG approach for an interactive application. Therefore, we investigate other approaches that make use of graphics hardware acceleration and are based on image-space calculations.

Planar Cutout. A simple image-space approach is based on the concept of clipping planes and allows for piecewise planar, convex cutouts according to *(R2)*. Each planar element of the cutout geometry is identified by a clipping plane. The exterior object is rendered n times where n is the number of different planes. In each rendering pass, the respective clipping plane is activated. Afterwards the interior objects are rendered in a single pass, with clipping planes being deactivated.

The advantages of the clipping-plane based method are its rather simple implementation and its support by virtually any graphics hardware. For example, client-defined clipping planes are already available in standard OpenGL 1.0. A drawback is the increase in rendering costs for multiple rendering passes—especially for more complex cutouts with several cutting planes. Another issue is the restriction to slick cutouts. Jittering boundaries according to *(R4)* are not possible. Since no explicit modeling of the cutout surface is implemented, the wall cannot be made visible *(R5)*.

Cutout via Stencil Test. The following screen-space technique exploits the stencil buffer and stencil test to represent the cutout geometry. The advantage of this method is the fact that jittering boundaries are supported. On the other hand, we restrict ourselves to convex exterior objects. Rappoport and Spitz[21] demonstrate that a related stencil-based approach can be used to make possible interactive Boolean operators for CSG.

Our algorithm is similar to implementation of shadow volumes by means of the stencil buffer[15]. The following extensions are needed for stencil-based cutouts. First, the algorithm has to affect the visibility of objects and thus their z values. Therefore, a mechanism to adjust depth values has to be incorporated. Second, the algorithm has to allow for front and back faces of the exterior convex object because the line-of-sight (from the camera) may have intersections with one front and one back face of the exterior object. The core algorithm is applied twice: once for front and once for back faces.

The details of the rendering algorithm are as follows. In the first step, the front face of the exterior object that should be cut is rendered to the depth buffer; the color buffer is masked out. Afterwards the front faces of the cutout geometry are rendered with the depth test being activated, but without changing the depth buffer entries. A stencil operator increases the stencil value by one each time a fragment passes the depth test. Similarly, the back faces of the cutout geometry are rendered without changing the depth buffer. This time the stencil value is decreased when a fragment passes the depth test. This ensures that pixels of the front face lying inside the cutout geometry have a stencil value of one. All other pixels have a stencil value of zero. Note that we have assumed that the camera is not located inside the cutout geometry. If the camera is within the cutout geometry, the stencil buffer needs to be initialized to one.

In the next step, the depth buffer is cleared and the back face of the exterior object is rendered to the depth buffer in the regions where the stencil value is greater than zero. Like in the second step, the front faces of the cutout geometry and then the back faces of the cutout geometry are rendered with depth test. The stencil value is increased for front faces passing the depth test and decreased for back faces passing the depth test. It is now possible to decide for each pixel if only the back face of the exterior object (stencil value one), both the back and front faces (stencil value zero) or neither (stencil value two) are visible. For the final step the depth buffer is cleared and then the back face of the exterior object is both rendered into depth and color buffers in regions where the stencil value is one. Afterwards the front face of the exterior object is rendered to depth and color buffers in those parts of the screen where the stencil value is zero. Finally, the interior objects are rendered without stencil test.

The advantage of this approach is that complex cutout geometries are supported. In addition, the number of rendering passes does not increase with the complexity of the cutout geometry. Stencil buffer and stencil test are already included in standard OpenGL 1.2. Therefore, the algorithm is widely supported by graphics hardware. Unfortunately, the structure of the exterior object is subject to an important restriction. The object has to be represented by a single convex surface without boundaries. The above algorithm makes explicit use of the fact that (at the most) one front and one back face is cut by a ray originating from the camera. However, many technical 3D data sets contain nested surfaces or explicitly represent all boundaries—both inside and outside. This means they can have more than just one front and back face intersecting the same line of sight. In this case the above algorithm fails because it is no longer guaranteed that all back faces of the exterior object lie behind the cutout geometry. Moreover, the cutout walls are not modeled *(R5)*.

Texture-Based Cutout. To overcome the restriction to convex exterior objects we present a new rendering algorithm that exploits texture mapping to represent the cutout geometry. The implementation requires programmable transform and lighting, per-fragment operations, and multi-textures.

First, we illustrate the basic idea of our approach by restricting ourselves to a single cutting plane. The scenario is depicted in Figure 2. Let us consider the required operations and tests to allow for a cut into a single triangle. The decision whether a fragment of the triangle lies inside the clipped half space or in the complement space is based on the signed Euclidean distance of the fragment from the plane. We define that fragments with a negative distance d are clipped and fragments with a positive value d pass. The signed distances are computed for each vertex and then interpolated across the triangle to obtain values for each fragment. The per-vertex distances can either be computed on the CPU or by a vertex program in the transform and lighting part of the rendering pipeline. The necessary parameters for the plane

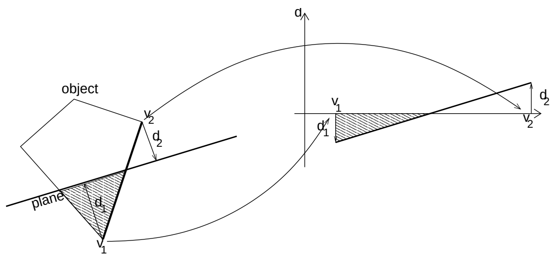

Figure 2: *Planar cutout based on a linearly interpolated signed distance.*

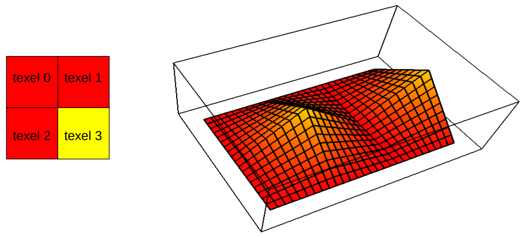

Figure 4: *A tiny 2×2 example texture used for sawtooth-shaped boundaries.*

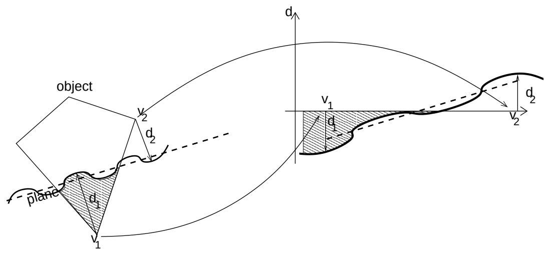

Figure 3: *Jittering cutout. A perturbation function displaces the original distances to the cutting plane.*

equation of the cutout can be provided by passing vertex program parameters.

First, the signed distance is stored in a texture coordinate. Texture coordinates are better suited than color-components for the following reasons: On most GPUs texture coordinates are implemented as floating point numbers, they are not restricted to the range $[0,1]$, and they have a much higher accuracy. Moreover, texture coordinates can be interpolated in a perspectively correct manner, i.e., a hyperbolic interpolation in screen space corresponds to a correct linear interpolation in object space.

The original signed distance d_{plane} can also be perturbed to allow for jittering cutouts. The idea is based on displacement mapping techniques[4]. Figure 3 illustrates the displacement of the signed distance to a cutting plane. The final distance is computed by fragment operations according to

$$d = d_{\text{plane}} + d_{\text{perturb}} \quad . \tag{1}$$

The perturbation d_{perturb} is stored in a 2D texture and is superimposed onto the original distance. While experimenting with different kind of perturbation textures we noticed that fractals, synthetic procedural textures used for clouds and virtual terrain height fields, and real terrain data produce good visual effects. In particular, a sawtooth-like boundary can be realized by a quite simple and memory friendly perturbation texture. Just a tiny 2×2 texture as illustrated in Figure 4 is needed. Thanks to texture repeat and bilinear texture interpolation a repeated falloff is generated which leads to the desired visual effect. Figure 1 (a) shows an example of a cutout illustration based on this 2×2 perturbation texture.

In the last step a fragment clipping operation has to be executed according to the corresponding distance value. If the value is below zero the fragment has to be clipped, otherwise kept. This can either be done through a texkill command or by setting the alpha value and using the alpha test.

So far only a single perturbed cutout plane is supported. The above texture-based algorithm can easily be extended to several cutout planes. A separate texture coordinate is used for each plane to store the respective signed distance. The same perturbation is applied to each distance value. An additional fragment operation determines the minimal absolute distance value.

The following approximation can be used to avoid several texture coordinates and the additional fragment operation. The minimal absolute distance value can also be computed per vertex and the according signed distance can be used as the only texture coordinate. This approximation yields correct results for most cases. If, however, the three different vertices of a triangle do not have the same closest plane, inaccuracies are introduced by interpolating signed distances that are attached to different planes. The smaller the triangle, the smaller is the possible error.

We show how the above texture-based cutout can be implemented on a great variety of graphic boards. First, a vertex program is enabled in order to compute the signed distances to the planes of the cutout geometry. We follow the above approximation and compute the minimal absolute distance value on a per-vertex basis. The signed distance to the closest plane is used as a texture coordinate. To implement the jittering cuting boundary different approaches can be used. On the Geforce 3 a texture shader program using three stages is utilized. In texture stage zero, the perturbation value is obtained by a lookup in the 2D perturbation texture. Stage one implements the shift of distance values according to Eq. 1. This shift is based on a computation of a dot product between two 3D vectors. The first vector originates from the RGB values from above perturbation texture. In this texture, the red channel is set to one, the green channel represents the value of the height field, the blue channel is set to zero. The second vector is given by the texture coordinates for stage one. On the ATI Radeon 8500 or Radeon 9700 a jittering is achieved by adding an offset to the texture coordinate of the current fragment. This offset can be looked up in a jittering texture.

The presented texture-based cutout algorithm meets the requirements *(R1)–(R4)*. The geometry of the interior and exterior objects is not subject to any restrictions. Since the algorithm can be mapped to graphics hardware, interactive frame rates are possible even for complex illustrations. The only drawback is the missing modeling of the cutout walls *(R5)*.

5. Breakaway Illustrations

In this section, a rendering algorithm that meets the requirements for breakaway illustrations is presented. We show how a computer-based process can fulfill the breakaway-specific rules *(R6)* and *(R7)*. The classification of objects as interior or exterior *(R1)* is provided by the same mechanism as in the previous section.

The basic idea is to clip away those parts of the surrounding object that would otherwise occlude the interior objects as seen from the camera's position. Therefore, this approach is intrinsically view-dependent and allows for *(R7)*. The other requirement is that only a single hole is cut into the exterior object *(R6)*. The convex hull of the interior objects is used as a basis for breakaway illustrations. Just enough is removed from the outside object to make this convex hull visible. The convex hull has two advantages. First, it contains all interior objects. If the convex hull is visible, all interior objects are visible. Second, the projection of the convex hull onto the image plane always yields a convex geometry and cannot contain any holes.

We propose the following algorithm for breakaway illustrations. In a preprocessing step, the convex hull of the interior objects is computed, for example, by the Quick Hull algorithm[1]. The convex hull is extended into all directions by some additional spatial offset. In this way, all interior objects are enclosed with a non-zero minimum distance to the hull. During the actual rendering process the extended convex hull serves as a virtual clipping object. Only those parts of surrounding objects that are not in front of the convex hull are rendered. This is achieved by using the foremost part of the convex hull as a clipping object. Finally, the interior objects are displayed.

The crucial point of the algorithm is the clipping at the foremost surface of the convex hull. A mechanism to clip away objects in front of an arbitrarily shaped object has to be employed. We use a clipping algorithm that is very similar to one of Diepstraten et al.'s[5] techniques for rendering transparent surfaces. Alternatively, Everitt's *depth-peeling*[9] could also be used. Both algorithms can be realized on a GeForce 3.

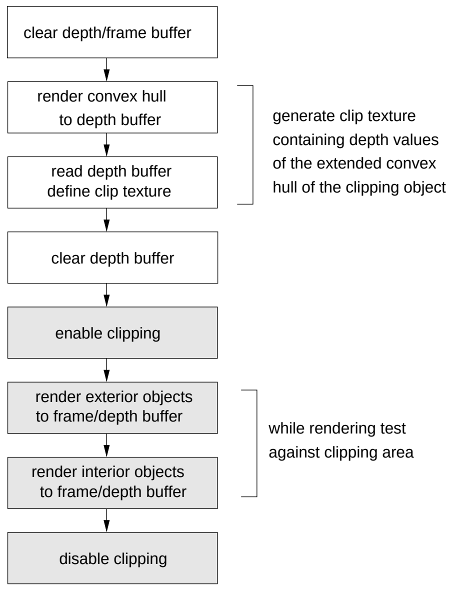
clear depth/frame buffer
render convex hull to depth buffer
read depth buffer define clip texture
generate clip texture containing depth values of the extended convex hull of the clipping object
clear depth buffer
enable clipping
render exterior objects to frame/depth buffer
render interior objects to frame/depth buffer
while rendering test against clipping area
disable clipping

Figure 5: *Rendering pipeline for the breakaway technique.*

Figure 5 shows the details of the rendering pipeline. In this case we use the terminology from the standard OpenGL specification and NVidia-specific extensions for the GeForce 3.

In the first four boxes, the clipping area in screen space is determined. First, the depth buffer is initialized with a depth of zero (which corresponds to the near clipping plane of the view frustum). Then, the extended convex hull is rendered twice into the depth buffer. The first rendering pass uses "greater" as logical operation for the depth test, the second rendering pass uses the standard "less" depth test. In this way, the depth buffer contains the depth values of the foremost parts of the convex hull. Furthermore, the depth buffer is still initialized with zero in the areas that are not covered by the convex hull. In the third box, the depth values are transferred into a high-resolution 2D texture (a HILO texture) that will serve as the clip texture. Then the depth buffer is cleared.

In the fifth box, a texture shader program is enabled to virtually clip away all fragments that have equal or smaller depth values than those given by the above clip texture. Essentially, this texture shader program replaces the z value of a fragment by $z - z_{\text{clip}}$, where z_{clip} represents the depth value stored in the clip texture. This texture shader program causes all fragments with $z < z_{\text{clip}}$ to be clipped away. For details of the texture program we refer to [5]. Then the exterior object is rendered into the frame and depth buffers; all parts in front of the convex hull are clipped away. Finally, the interior objects are displayed and the texture shader program is reset to the standard configuration.

This algorithm can take into account more than one "cluster" of interior objects by computing several corresponding convex hulls and rendering them into the depth buffer in step 2 of the rendering pipeline. It is also possible to allow for surrounding objects with boundary surfaces of finite thickness. Here, separate clipping objects have to be defined for the front and for the back face of the boundary. By choosing a smaller clipping geometry for the back face, we can imitate the effect of cutting through an object of finite wall thickness. In this way, requirement *(R5)* for visible walls can be met in parts.

6. Implementation and Results

Our implementation of cutaway illustrations is based on OpenGL and on NVidia-specific extensions for the GeForce 3. User-interaction and the management of rendering contexts in our C++ application are handled by GLUT.

We have implemented several different NPR styles to demonstrate that our cutaway processes are independent of the rendering style. Figures 6 (a)–(c) show a cutout illustration of the same engine block with three different rendering styles. Additionally a 512×512 shadow map is used to simulate shadow casting inside the interior parts. In Figure 6 (a), we employ silhouette rendering with a toon-shading technique described by Lake et al.[16] Specular lighting is added to the original diffuse toon-like lighting. The diffuse and specular terms are used to access a 2D texture containing the final color. The silhouettes are generated by using a hardware-based method according to Gooch et al.[12] The silhouettes at cutting boundaries cannot be created by the original approach because the boundaries are not explicitly modeled as a triangular mesh. Therefore, the criterion for a silhouette—an edge connecting a front with a back face—is not valid at a boundary. This problem can be overcome by using an idea also described by Gooch et al.[12] Figure 6 (b) uses cool/warm tone shading as described by Gooch et al.[11] The shading model is implemented as a vertex program and provides per-vertex lighting. Finally, Figure 6 (c) uses a real-time layered-stroke texture approach described by Freudenberg et al.[10] with per-vertex lighting. Black line silhouettes are rendered according to Gooch et al.[12] All example images for the cutout illustrations are based on a synthetic heightfield texture to visualize a jittering boundary.

The same toon shading, cool/warm shading, and layered-stroke techniques are also used in breakaway illustrations of a curved conduit in Figures 6 (d)–(f). It can be clearly seen that the different rendering styles produce images of different visual qualities. For both cutaway techniques, the toon and cool/warm shading produce similar, convincing results. In contrast to these two rendering styles, the stipple images are—in our opinion—slightly unsatisfactory as they do not engender a good contrast between interior, exterior, and wall surfaces.

Table 1 shows performance measurements for both approaches with the different rendering styles. All tests were carried out on a Windows XP PC with AMD Athlon 1533Mhz CPU and GeForce 4 Ti 4600. The test scene is depicted in Figures 6 (a)–(c) and contains 145,113 triangles. Note that in each test silhouette lines are rendered as well and no special data structures like vertex arrays or displaylists were used in the measurements. We tested both cutaway techniques with three different rendering styles and on two different viewport sizes of 512^2 and 1024^2. For comparison, we included the rendering times for each style without applying cutaway methods. As it can be seen in Table 1, the cutout technique has not much influence on the final rendering time of any rendering style. At first glance this is quite suprising as at least three texture stages are needed to achieve a jittering cutout on the Geforce 3. A possible explanation could be that only very small textures which seem to fit perfectly into the texture cache are used. Probably, rendering performance for this test scene is limited by other factors such as the vertex pipeline. For the larger viewport size, the breakaway technique looses nearly half of its original performance. This might be due to the double readback from depth buffer for the convex hulls of the interior objects and the usage of large clipping textures which might enforce a lot of texture cache misses. Cache misses are a probable reason for the overall lower performance of the layered-stroke texture rendering style because a rather large texture (256^2 texels) is used compared to the toon-shading texture (8^2 texels).

Table 1: *Performance measurements in frames per second. The test model is illustrated in Figure 6(a)-(c) and contains 145,153 triangles.*

render style	cutaway technique	viewport size 512^2	viewport size 1024^2
cool/warm	none	6.11	6.10
cool/warm	cutout	6.07	5.69
cool/warm	breakaway	5.67	3.27
toon shading	none	6.12	6.10
toon shading	cutout	5.75	5.32
toon shading	breakaway	5.52	3.30
stroke textures	none	6.07	6.04
stroke textures	cutout	5.74	5.32
stroke textures	breakaway	4.73	2.81

7. Conclusion

In this paper, we have presented a small and effective set of rules for computer-based renderings of cutaway illustrations. One class of rules leads to cutout drawings, which are most appropriate for scenes with rather large interior objects.

The other class of rules is used for breakaway illustrations, which are suitable for scenes with smaller, densely packed interior objects. We have presented hardware-based methods for both cutout and breakaway drawings in order to achieve interactive frame rates. Even though our rules make possible a completely automatic generation of quite reasonable cutaway illustrations, interactivity is still very useful for a fine adjustment of the initial parameters. An advantage of our rules is the small number of parameters that effectively control the visual appearance of the drawings. Finally, cutaway techniques can be readily combined with existing non-photorealistic rendering styles, such as silhouette rendering, cool/warm tone shading, or pen-and-ink illustrations. One problem that cannot be addressed with our techniques is the visual appearance of the wall itself. Exploiting the increased functionality of future GPUs might help to address this problem.

Acknowledgments

We would like to thank the anonymous reviewers for helpful comments to improve the paper.

References

1. B. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. Technical Report GCG53, The Geometry Center MN, 1993.
2. D. Blythe. SIGGRAPH 1999 Course: Advanced Graphics Programming Techniques Using OpenGL, 1999.
3. D. Bourguignon, M.-P. Cani, and G. Drettakis. Drawing for illustration and annotation in 3D. *Computer Graphics Forum*, 20(3):114–122, 2001.
4. R. L. Cook. Shade trees. *Computer graphics (Proceedings of ACM SIGGRAPH 84)*, 18(3):223–231, 1984.
5. J. Diepstraten, D. Weiskopf, and T. Ertl. Transparency in technical illustrations. *Computer Graphics Forum*, 21(3):317–325, 2002.
6. D. Dooley and M. F. Cohen. Automatic illustration of 3D geometric models: Lines. *Computer graphics (Proceedings of ACM SIGGRAPH 90)*, 24(2):77–82, 1990.
7. D. Dooley and M. F. Cohen. Automatic illustration of 3D geometric models: Surfaces. *IEEE Computer Graphics and Applications*, 13(2):307–314, 1990.
8. R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.
9. C. Everitt. Interactive order-independent transparency. White paper, NVidia, 2001.
10. B. Freudenberg, M. Masuch, and T. Strothotte. Real-time halftoning: A primitive for non-photorealistic shading. In *Proceedings of Eurographics Workshop on Rendering*, pages 227–231, 2002.
11. A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of ACM SIGGRAPH 98*, pages 101–108, July 1998.
12. B. Gooch, P.-P. Sloan, A. Gooch, P. Shirley, and R. Riesenfeld. Interactive technical illustration. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999.
13. S. Gottschalk, M. C. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. In *Proceedings of ACM SIGGRAPH 96*, pages 171–180, 1996.
14. J. Hamel, S. Schlechtweg, and T. Strothotte. An approach to visualizing transparency in computer-generated line drawings. In *Proceedings of IEEE Information Visualization 1998*, pages 151–156, 1998.
15. M. Kilgard. Creating reflections and shadows using stencil buffers (NVidia technical demonstration). Game Developer Conference, 1999.
16. A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, pages 13–20, 2000.
17. J. Martin. *Technical Illustration: Material, Methods, and Techniques*, volume 1. Macdonald and Co, 1989.
18. A. Mohr, E. Bakke, A. Gardner, C. Herrman, and S. Dutcher. NPR Quake. http://www.cs.wisc.edu/graphics/Gallery/NPRQuake, 2002.
19. F. S. Nooruddin and G. Turk. Interior/exterior classification of polygonal models. In *Proceedings of IEEE Visualization 2000*, pages 415–422, 2000.
20. E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *Proceedings of ACM SIGGRAPH 2001*, pages 579–584, 2001.
21. A. Rappoport and S. Spitz. Interactive Boolean operations for conceptual design for 3-D solids. In *Proceedings of ACM SIGGRAPH 97*, pages 269–278, 1997.
22. R. Raskar. Hardware support for non-photorealistic rendering. *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 41–47, 2001.
23. M. Salisbury, S. Anderson, R. Barzel, and D. Salesin. Interactive pen and ink illustration. In *Proceedings of ACM SIGGRAPH 94*, pages 101–108, 1994.
24. D. D. Seligmann and S. Feiner. Automated generation of intent-based 3D illustrations. *Computer Graphics (Proceedings of ACM SIGGRAPH 91)*, 25(4):123–132, 1991.
25. D. D. Seligmann and S. Feiner. Supporting interactivity in automated 3D illustrations. In *Proceedings of the 1st International Conference on Intelligent User Interfaces*, pages 37–44. ACM Press, 1993.
26. T. Thomas. *Technical Illustration*. McGraw-Hill, second edition, 1968.
27. G. Winkenbach and D. H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of ACM SIGGRAPH 94*, pages 91–100, 1994.

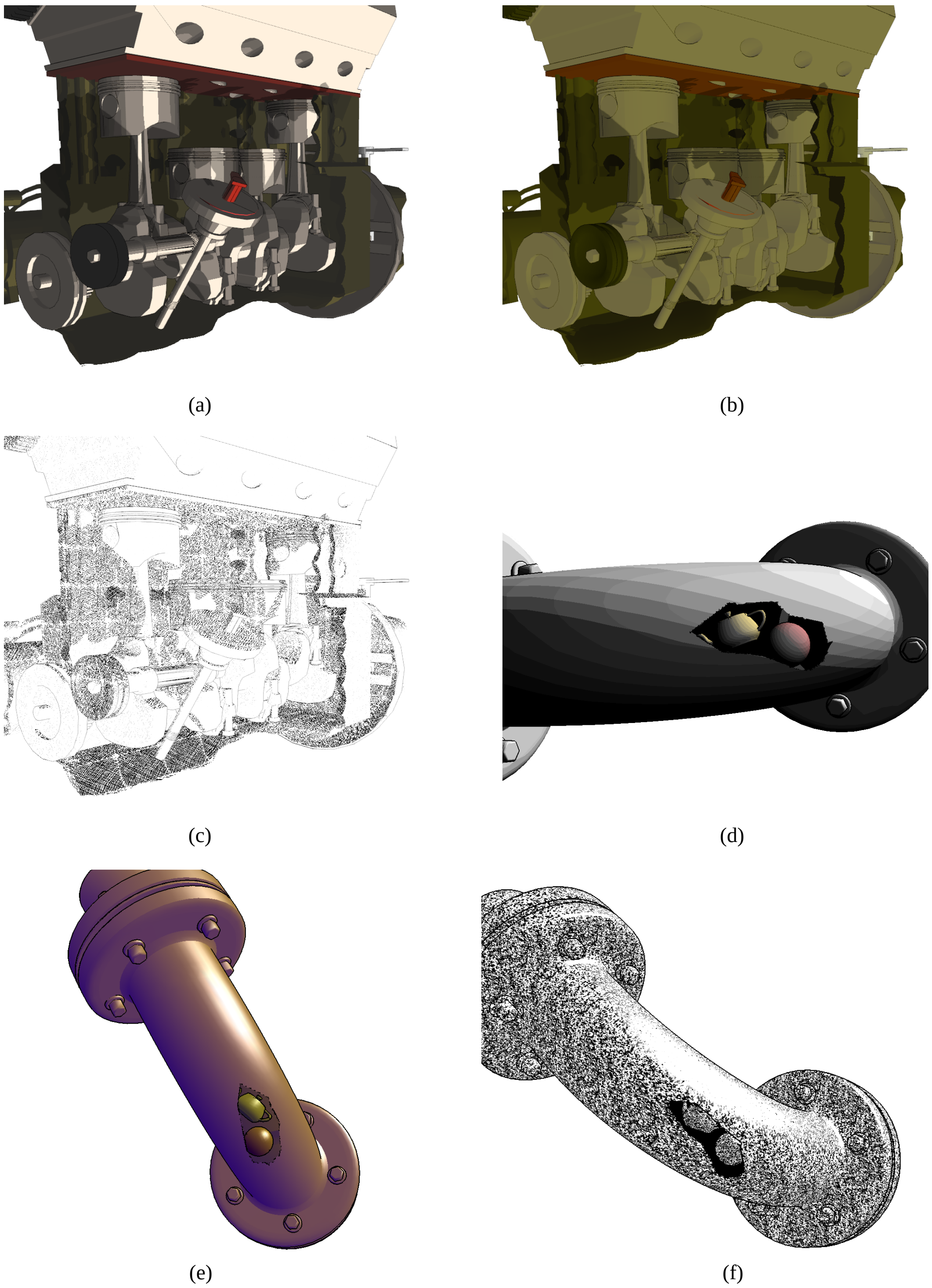

Figure 6: *Cutout and breakaway illustrations with different rendering styles. Image (a) shows a part of an engine block with toon shading and silhouette rendering, (b) shows the same scene using cool/warm tone shading with black silhouette lines, (c) illustrates the same scene using layered-stroke textures. Images (d), (e), and (f) show the breakaway technique for the example of a curved conduit; the same rendering styles are applied as in (a)–(c).*